Anduril User Guide
February 27, 2015

Kristian Ovaska
Ping Chen
Marko Laakso
Ville Rantanen
Riku Louhimo
Sirkku Karinen
Javier Núñez Fontarnau
Vladimir Rogojin
Roman Sirokov

Contact: kristian.ovaska@helsinki.fi
Biomedicum Helsinki, Finland

Contents

I Anduril for End Users 1

1 Introduction to Anduril 1
- 1.1 Component model . . . 2
  - 1.1.1 Type parameters . . . 4
  - 1.1.2 Resource bundles . . . 4
- 1.2 Workflows . . . 4
  - 1.2.1 Composite components . . . 5
  - 1.2.2 Conditional branches . . . 5
- 1.3 Workflow execution . . . 6
- 1.4 Component and workflow quality control . . . 7

2 Installation and requirements 9
- 2.1 VirtualBox image installation . . . 9
- 2.2 Debian package installation . . . 13
- 2.3 Binary package installation . . . 13

3 Using Anduril 16
- 3.1 Eclipse interface . . . 16
  - 3.1.1 Eclipse plugin installation . . . 16
  - 3.1.2 Configuring the plugin . . . 17
  - 3.1.3 Constructing workflows . . . 17
  - 3.1.4 Executing a workflow . . . 19
  - 3.1.5 Remote execution over SSH . . . 20
- 3.2 Command line interface . . . 21
  - 3.2.1 Executing a workflow . . . 21
  - 3.2.2 Executing a single component . . . 24
  - 3.2.3 Executing component test cases . . . 25
  - 3.2.4 Executing a workflow with the #! runner . . . 26
  - 3.2.5 Advanced command line usage and debugging . . . 26
- 3.3 Apache Ant interface . . . 27
  - 3.3.1 Executing a workflow . . . 27
- 3.4 Anduril graphical user interface . . . 29
- 3.5 Browsing the execution folder . . . 30
- 3.6 Isolated execution of a component instance in a workflow for debugging 32
- 3.7 Modes of execution . . . 33
  - 3.7.1 Slurm . . . 33
  - 3.7.2 Prefix . . . 34

4 Life sciences analysis 35
- 4.1 Data types and file formats . . . 35
- 4.2 Components . . . 36
  - 4.2.1 Basic INPUT and OUTPUT . . . 36

4.2.2 General purpose processing . . . 36
4.2.3 Annotation using databases . . . 37
4.2.4 Statistics, data mining and plotting . . . 38
4.2.5 Analyzing expression microarrays . . . 38
4.2.6 Analysing single nucleotide polymorphisms . . . 39
4.2.7 Exon array analysis . . . 39
4.2.8 Gene Ontology and pathway analysis . . . 40
4.2.9 Report generation . . . 40
4.3 Worked Examples . . . 41
4.3.1 Getting started: random matrices . . . 41
4.3.2 Two-channel Agilent arrayCGH microarrays . . . 43
4.3.3 Integrate fold change information with protein-protein interaction network . . . 48

5 Workflow construction using AndurilScript **52**
5.1 Concepts . . . 53
5.2 Basic syntax . . . 55
5.3 Placing component instances on a workflow . . . 57
5.4 Defining composite components: `function` . . . 61
5.5 Including other files: `include` . . . 62
5.6 Static conditional processing: `if` – `else` . . . 63
5.7 Dynamic conditional branches: `switch` – `case` . . . 63
5.8 Native functions . . . 65
5.8.1 Generic functions . . . 65
5.8.2 File functions . . . 68
5.8.3 String functions . . . 69
5.8.4 Numeric functions . . . 70
5.9 Looping over iterators: `for` . . . 71
5.10 Dynamic `for`-loop and `include` statement . . . 75
5.11 Array data types . . . 76
5.12 Executing components on remote hosts . . . 78
5.12.1 Remote host configuration file . . . 79
5.12.2 Example . . . 80

II Anduril for Developers **83**

6 Implementing components **83**
6.1 Directory structure . . . 83
6.2 Descriptor XML files . . . 83
6.2.1 Launchers . . . 84
6.2.2 Branch components . . . 88
6.2.3 Type parameters . . . 88
6.2.4 Declaring array ports . . . 89
6.2.5 Multifiles . . . 89
6.2.6 Defining component requirements . . . 90

6.3 Component execution . . . 91
6.3.1 Command files . . . 92
6.3.2 Array data type implementation . . . 92
6.4 Component frameworks . . . 93
6.4.1 Framework for R . . . 93
6.4.2 Framework for Java . . . 94
6.4.3 Framework for Matlab . . . 94
6.4.4 Framework for bash . . . 94
6.4.5 Framework for Python . . . 94
6.4.6 Framework for Lua . . . 94
6.5 Component test cases . . . 95
6.6 Example component: adding matrices . . . 96
6.7 Guidelines for designing components . . . 97
6.8 Implementing support for new programming languages . . . 99

7 Defining port data types 101

8 Resource bundles 102
8.1 Bundle definition XML files . . . 103
8.2 Category definition XML files . . . 103
8.3 Workflow-level test cases . . . 104
8.4 Composite component libraries . . . 104

9 Integrating Anduril into Java programs 105

Glossary

Annotation for Component Special parameter that modifies the execution logic of component instances. 58

Annotation for Port Special parameter that modifies the attributes of a component instance port. 60

Annotations Collection A record that maps annotation labels onto valid values. 60

Array data type Data format for input and output ports that consists of an ordered collection of (key, file) pairs. 76, 89, 92

Branch Dynamic conditional branch on a workflow where alternative execution routes are selected for execution. 5, 63, 88

Component Reusable executable, which implements a part of a workflow, following a standard Anduril interface. 2, 24, 83

Component framework (*Developers*) Programming-language specific convenience library that implements common component tasks. 93, 99

Component instance A component placed on a workflow with values for simple parameters and port connections to other instances. 4, 57

Composite component Sub-workflow encapsulating several other components. Used like a regular component. 5, 61, 104

Input port Specifies one data item (file or directory) that a component reads in. 2

Multifile A kind of a port data type that consists of a primary file and optional auxiliary files in the same directory with the same basename but different file extension. 89, 101

Output port Specifies one data item (file or directory) that a component produces. 2, 57

Port connection Connection from an output port to an input port. Port types must match. 4

Port data type Standardized data format for input and output ports. 101

Resource bundle Directory structure that contains component implementations and related data types. 4, 102

Simple parameter String, number or boolean value that is passed to a component. 2, 57

Test case Set of input files, expected output files, and configuration files that is used to test whether a component or workflow works properly. 7, 25, 95, 104

Type parameter Placeholder for a concrete port type. Used for components that preserve the data format of their inputs. 4, 88

Workflow Component instances wired together using port connections. Executed by the Anduril workflow engine. 4, 19, 21, 27, 52

Part I
Anduril for End Users

1 Introduction to Anduril

Anduril (ANalysis of Data Using Rapid Integration of aLgorithms) is a component-based workflow framework for bioinformatics and other scientific data analysis. Anduril aims to enable systematic, scalable and flexible data analysis. Anduril is most suitable for users who have elementary programming experience, such as bioinformaticians.

A *workflow* is a series of processing steps connected together so that the output of one step is used as the input of another. Processing steps implement data analysis tasks such as data importing, statistical tests and report generation. In Anduril, processing steps are implemented using *components*, which are reusable executable code that can be written in any programming language. Components are wired together into a workflow, or a component network, that is executed by the Anduril workflow engine. Workflow configuration is done using an easy-to-learn yet powerful scripting language, AndurilScript (see Section 5). Workflow configuration and execution can be done from Eclipse, a popular multipurpose GUI, or from the command line (see Section 3).

The Anduril architecture can be divided into three levels, illustrated in Figure 1. The core level provides a workflow engine and a few central components, but is not specific to any application area. On the second level, components collections, or application-area specific frameworks, provide components for a particular area. They enable Anduril to perform real-world tasks such as analyze biological data. Finally, analysis projects are implemented as workflows that use the components defined by component collections. Components are reused between workflows. Third parties can also implement new components.

This manual explains the core level and gives and overview on life sciences related components bundled with Anduril (Section 4), but the Anduril framework is more general and can be adopted to other data analysis tasks as well.

Anduril is distributed under an open source licence and is available for multiple platforms including Linux and Windows. The web site of Anduril is at
http://csbi.ltdk.helsinki.fi/anduril/.

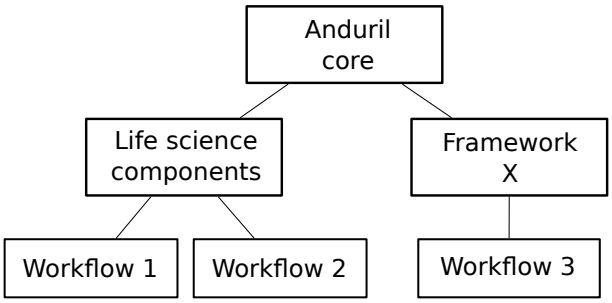


Figure 1: Anduril high-level architecture. The core provides facilities for executing workflows, generating component manual pages and other general tasks. Component collections (second level) enable Anduril to perform real-world tasks. There can be several such collections. Invididual workflows utilize the component collections.

1.1 Component model

A component is an executable which reads data from input ports and writes processed results into output ports. Input and output *ports* represent files or directories and may contain numeric matrices, binary files or directory structures, for example. There can be any number of input and output ports. An input port may be marked as optional in which case the corresponding data item may be missing. Each component has a generated HTML manual page that describes the ports, parameters and other features of the component. The component model is programming language independent because the only requirement is the ability to read and write files.

In addition to ports, components have simple parameters such as strings, numbers and truth values (Booleans). Parameters are used to tweak component execution. A statistical test component might have a numeric parameter for the p-value threshold, for instance. Simple parameters are never optional, *i.e.*, they must always be provided. However, they typically have default values.

Ports and simple parameters have associated data types which specify the kind of values ports and parameters might obtain. Port types correspond to file formats. The port type system is extensible, *i.e.*, third-party developers can define new types. Existing types include CSV files, numeric matrices, PDFs and numerous others. Like components, port types are documented in the manual pages generated by Anduril. Port types have a hierarchical structure so that, for instance, `Matrix` (a numeric matrix stored as a CSV file) is a subtype of `CSV`. A component that takes in a `CSV` file will also accept a `Matrix`. The types for simple parameters are fixed and are listed in Table 1.

Figure 2 shows the manual page of an example component, AddMatrix. This simplified

Type ID	Description
`int`	Integer
`float`	Real number
`string`	Character sequence
`boolean`	Truth value

Table 1: Data types for simple parameters.

Component

AddMatrix

Compute the sum of two or three matrices. Add a constant bias to the result.

Version 1.0
Categories arithmetic
Requires R

Inputs

Name	Type	Mandatory	Description
`m1`	Matrix	Mandatory	Input matrix 1.
`m2`	Matrix	Mandatory	Input matrix 2.
`m3`	Matrix	Optional	Input matrix 3.

Outputs

Name	Type	Description
`sum`	Matrix	Sum of matrices m1, m2 and m3 (if defined), plus bias.

Parameters

Name	Type	Default	Description
`bias`	float	0	A bias that is added to all cells of the output matrix.

Figure 2: Manual page of AddMatrix, a simple example component. The page is generated by Anduril from the component interface definition XML file. The XML file is shown in Figure 18 and R source code in Figure 19.

component computes the sum of two or three numeric matrices. It has only one output port, but in general components may have several output ports. AddMatrix has one numeric parameter (`bias`) that has a default value of zero.

1.1.1 Type parameters

Components may have *type parameters* so that the type of a port is not a concrete type but a generic one. Such a component is called a *generic* component. When the component is placed on a workflow, type parameters are automatically assigned concrete types based on the connections of the generic component. Type parameters are useful for filter components that modify the contents of files but preserve the file format. A trivial filter would be a copy component that copies the contents of input files to output files. The input and output ports would have a generic type since the component can act on any file type and preserves the type. For details, see Section 6.2.3. The generics mechanism is largely transparent to the user.

1.1.2 Resource bundles

Related components and their data types can be packaged together into a *resource bundle*, or bundle for short. Bundles have a well-defined directory structure that contains component implementations; data type definitions; test cases for both component and the bundle in general; and libraries shared by components. Anduril comes prepackaged with a bundle of life science related components, but additional bundles can be installed. See Section 8 for details on bundle file structure.

1.2 Workflows

A workflow is created by placing components into a component network where the output port of one component is connected to the input of another. A component placed in a workflow is called a *component instance*; there may be several instances of the same component in one workflow. Component instances connected through ports have a dependency on each other: the source instance must be executed before the target instance. Figure 3 shows a schematic of a workflow. Workflows are created and configured using a high-level script language, AndurilScript (Section 5). See Figure 13 (Section 4.3) for an example AndurilScript program and its automatically generated visualization. Table 6 (Section 5) contains a ”cheat sheet” of AndurilScript.

The types of input and output ports in a port connection must be compatible; the workflow engine reports an error otherwise. A mandatory (non-optional) input port must have exactly one incoming connection, while an optional input port may have no connections or one connection. An output port may be connected to any number of input ports.

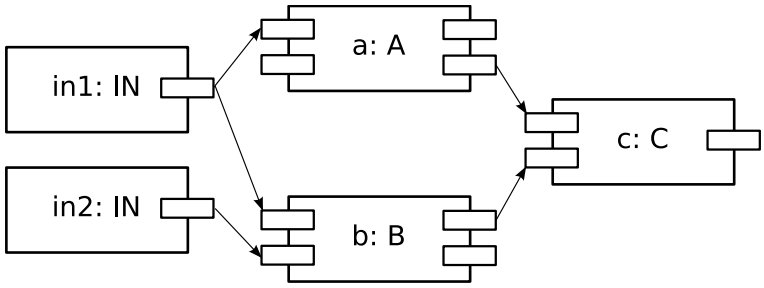


Figure 3: A workflow composed of five component instances. The labels show names of component instances and names of components. For example, `in1` is an instance of the `IN` component. Another instance of `IN` is `in2`. Input ports are on the left and output ports on the right. Arrows represent port connections. The second output port of `a` is connected to the first input port of `c`. The sole output port of `in1` is connected to two input ports. Port names and simple parameters are omitted for clarity.

1.2.1 Composite components

To help in creating complex workflows, *composite components* can be used to break a workflow into hierarchical sub-workflows. A composite component comprises other components that together define the sub-workflow, *i.e.*, composition. Like regular components, a composite component may have input and output ports and simple parameters. This is similar to how function definitions are used to break a program into manageable pieces in programming languages. Indeed, in AndurilScript composite components are defined using a function-like syntax as seen in Section 5.4. A composite component can be instantiated several times in a workflow, like regular components. Different instances may be composed of different sets of components, since the definition of a composite component may contain conditional processing based on parameters (if-statements, Section 5.6).

1.2.2 Conditional branches

Conditional branches can be used to dynamically select alternative execution routes in the workflow. Branches are able to use dynamic results produced by other components and are therefore a powerful mechanism to add flexibility to workflows.

A branch is composed of three elements: a branch component, two or more alternative routes and a join component that ends the branch. When the branch component is executed, it enables one or more of the alternatives. At least one alternative must be enabled. Each alternative is composed of components that are executed if the alternative is enabled. End results of the alternative routes are passed to the join component, which

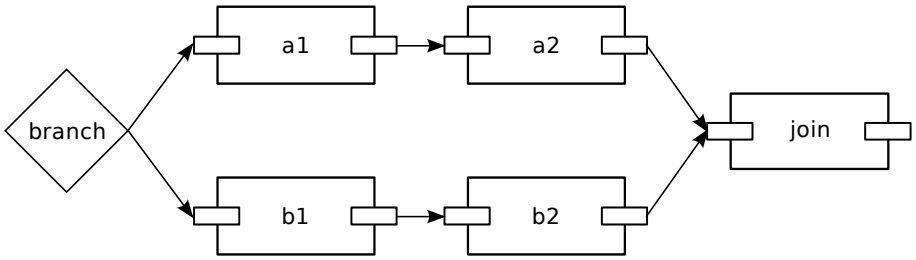


Figure 4: Conditional branch. There are two alternative routes, {`a1`, `a2`} and {`b1`, `b2`} that can be independently enabled by the branch component. At least one of the alternatives must be enabled. The branch is ended by the `join` component. Although not shown, the branch component can also have input and output ports.

combines them into a final result. Figure 4 shows a branched workflow. See Section 5.7 for details on setting up conditional branching.

1.3 Workflow execution

The workflow is executed by the Anduril workflow engine by invoking component instances. Each component is executed in a new parallel process. After processing its input and writing output files, the component process exits and signals a success/failure status to the engine, enabling components that depend on the output of the present component to be executed. For details on single component execution, see Section 6.3.

Execution starts from the ”source” component instances, that do not have input dependencies, and propagates towards the end point component instances that do not have output dependencies. In Figure 3, `in1` and `in2` are sources and `c` is an end point. In general, there may be several end points.

Components that do not depend on each other are automatically executed in parallel. This allows for taking advantage of multi-core CPUs without requiring the user to take care of details such as process creation and synchronization. The maximum number of parallel threads can be configured.

Component outputs are stored on disk in an *execution directory*. The workflow engine assigns a unique file (or directory) to each output port of each component instance. Only the owner instance is allowed to write to that file. The file associated to an input port is defined by the port connection: the input file is an output file of another component. Hence, several component instances can read the same file when an output port is connected to several input ports.

Partial re-execution When a workflow is executed several times, only those component instances, whose configuration has changed, are executed. This greatly saves time, as costly preprocessing steps do not have to be re-executed except when their configuration is changed. When necessary, the user has an option to force execution of selected or all component instances (see Section 3).

The outputs of components are cached on disk. Also, configuration settings such as simple parameters are stored. When a workflow is executed again, the workflow engine automatically detects components whose configuration has changed after the previous run. A component instance is considered as changed if its simple parameters, input connections, version number or timestamps of input files have changed. Only changed component instances and those that depend on changed instances are re-executed on subsequent runs. To save disk space, it is possible to disable output caching for individual or all components (Section 5.3 and `--min-space` command line switch).

As a consequence of ”smart” re-execution, each component instance must be executed at most once and the workflow network must not contain cycles. The engine ensures that there are no cycles in the workflow.

1.4 Component and workflow quality control

Components and workflows may both have test cases that aim to ensure the components are working properly in the environment of the user. Test cases are also useful during the development of new components and modification of existing ones. Test cases are executed using the Anduril core.

Test cases of components are based on the component interface. Each test case provides input files and may set simple parameters. A test case expecting successful execution provides expected output files. Anduril executes the component using input files and parameters given by the test case and compares actual results to expected results. The test case is passed if execution finished successfully and actual outputs agree with expected outputs. For some file types, such as binary files containing time stamps, it is difficult to compare actual results to expected results. For such file types, the expected output may be omitted. A test case may also expect failed execution, in which case there are no expected outputs.

Workflow tests exercise the integration interfaces between components and aim to ensure that components work together. A workflow test is composed of a workflow configuration, input files and expected outputs of selected components. Analogously to component testing, Anduril executes the workflow and compares actual results to

expected results. Only selected outputs of selected components are compared. Like component tests, workflow tests may also expect failed execution.

2 Installation and requirements

The core Anduril engine is written in Java and components are written in a variety of programming languages, including Java and R. Components may also have dependencies on third-part libraries, such as Bioconductor. The dependencies of components are listed on their manual pages. The Anduril core depends only on Java and has been tested on Linux and Windows. Most components also use portable languages and are reasonably platform-independent. You can verify that a component is working on your platform by executing component tests.

Anduril downloads are available at http://csbi.ltdk.helsinki.fi/anduril/

Anduril is distributed as several alternative package types. All installation types include the Anduril core engine; source code for core and most components; component repository; and documentation including the User Guide, component and data type manual pages and technical Java API documentation. Some third-party libraries are bundled with Anduril and do not need to be installed explicitly; others are not bundled and may need to be installed, depending on the distribution type. The distribution types are:

- Virtual Ubuntu installation for Oracle VirtualBox (see http://www.virtualbox.org/). This is a disk image that contains an installation of Ubuntu Linux that can be invoked on any operating system supported by VirtualBox, including Windows, Linux and MacOS. The installation includes Java, R, Bioconductor libraries, LaTeX and other component dependencies pre-configured. This is the most convenient way to get started with Anduril.
- Debian package for Linux. This is meant for Linux systems (such as Debian and Ubuntu) that are able to install Debian packages. The package installs Java, R, LaTeX and other dependencies along with the Anduril core.
- Binary package. The package contains the Anduril system but no external dependencies such as Java or R.

2.1 VirtualBox image installation

The Oracle VirtualBox image is a virtual Xubuntu Linux installation (http://www.xubuntu.org/) that comes prepackaged with Anduril and various external dependencies, including Java, R and Bioconductor libraries. The virtual Linux installation can be invoked on any operating system supported by VirtualBox, including Windows, Linux and MacOS.

1. Download and install VirtualBox from http://www.virtualbox.org/. Either the binary version or the Open Source Edition can be used.

2. Download the Linux image from http://csbi.ltdk.helsinki.fi/anduril/. If the image is archived, unpack it. The final image should have the file extension `vdi`.
3. Launch VirtualBox.
4. Create a new virtual machine using "New" (Figure 5, 1). For "Operating System", select Linux (Figure 5, 2). The virtual machine should have at least 512 MB main memory. It is recommended to use more memory (e.g., 2 GB) if enough physical memory is available. It is not recommended to use more that half of physical memory for the virtual machine.
5. In Hard Disk configuration, select an "Existing" hard disk image (Figure 5, 3). In "Virtual Media Manager" (Figure 5, 4), select "Add" to import the downloaded hard disk image (`vdi` file).
6. Virtual machine configuration is now finished. To launch the virtual machine, select "Start" from the VirtualBox main window.

The virtual Xubuntu Linux is used like a regular desktop Linux. See Figure 6 for a screenshot of the installation. To use the command line interface (CLI) of Anduril, double-click "Terminal" and type `anduril`. See Section 3.2 for details on CLI usage. You may need type a password for some operations; the user name and password are both `anduril` on the virtual machine.

To enable exchanging data between the host and virtual machines, shared folders can be used. The contents of shared folders are visible on both the host and the virtual machine. While the virtual machine is running, select "Devices" $\rightarrow$ "Shared Folders" from the VirtualBox menu and add a shared folder. For easy operation, create a shared folder named `data`. This standard folder can be mounted on the virtual machine using the "Mount data" shortcut on the desktop. The folder is visible as $\sim$`/data` on the virtual machine. Alternatively, to manually mount a shared folder in the virtual machine, open Terminal and type

`sudo mount.vboxsf -o uid=1000,gid=1000,exec SHARENAME MOUNTPOINT`

where `SHARENAME` is the name of the shared folder you created and `MOUNTPOINT` is an existing directory in the virtual machine.

It is possible to connect from the host to the client using SSH. This is done by executing the following three commands (without extra newlines) on the host machine and starting the virtual machine again.

```
VBoxManage setextradata GUESTNAME
  "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/HostPort" 2222
VBoxManage setextradata GUESTNAME
  "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/GuestPort" 22
VBoxManage setextradata GUESTNAME
  "VBoxInternal/Devices/pcnet/0/LUN#0/Config/ssh/Protocol" TCP
```

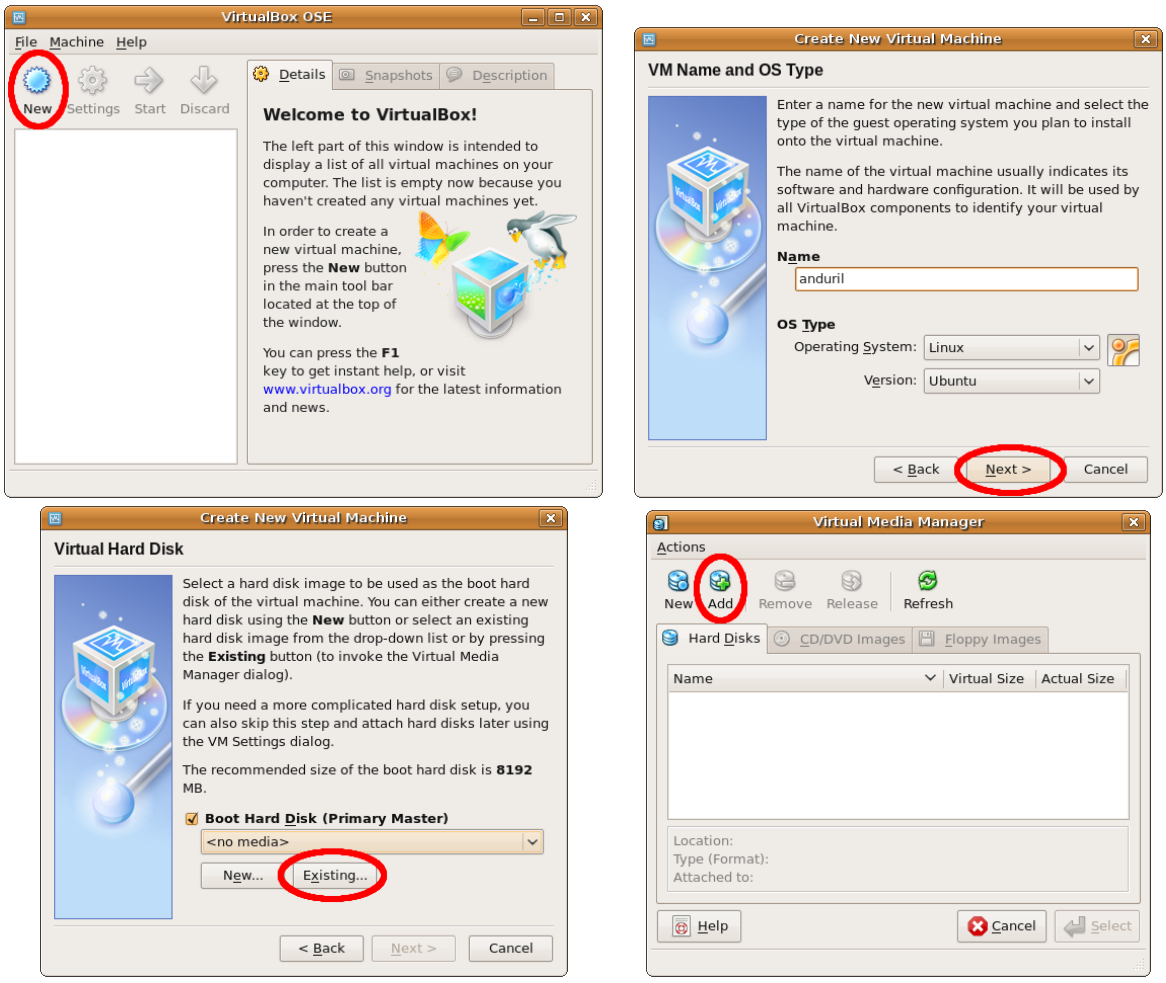

VirtualBox OSE
File Machine Help
New Settings Start Discard
Details Snapshots Description
Welcome to VirtualBox!
The left part of this window is intended to display a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.
In order to create a new virtual machine, press the New button in the main tool bar located at the top of the window.
You can press the F1 key to get instant help, or visit www.virtualbox.org for the latest information and news.
Create New Virtual Machine
VM Name and OS Type
Enter a name for the new virtual machine and select the type of the guest operating system you plan to install onto the virtual machine.
The name of the virtual machine usually indicates its software and hardware configuration. It will be used by all VirtualBox components to identify your virtual machine.
Name
anduril
OS Type
Operating System: Linux
Version: Ubuntu
< Back Next > Cancel
Create New Virtual Machine
Virtual Hard Disk
Select a hard disk image to be used as the boot hard disk of the virtual machine. You can either create a new hard disk using the New button or select an existing hard disk image from the drop-down list or by pressing the Existing button (to invoke the Virtual Media Manager dialog).
If you need a more complicated hard disk setup, you can also skip this step and attach hard disks later using the VM Settings dialog.
The recommended size of the boot hard disk is 8192 MB.
Boot Hard Disk (Primary Master)
<no media>
New... Existing...
< Back Next > Cancel
Virtual Media Manager
Actions
New Add Remove Release Refresh
Hard Disks CD/DVD Images Floppy Images
Name Virtual Size Actual Size
Location:
Type (Format):
Attached to:
Help Cancel Select

Figure 5: VirtualBox image installation.

Here, GUESTNAME is the name of the virtual machine. Now, TCP port 2222 on the host machine is forwarded to TCP port 22 (SSH) on the virtual machine. The SSH connection can be initiated from the host machine using the command `ssh -p 2222 -l anduril localhost`. The password is `anduril`.

Resolution of the virtual desktop can be changed in ”Applications” → ”Settings” → ”Settings manager” → ”Display”.

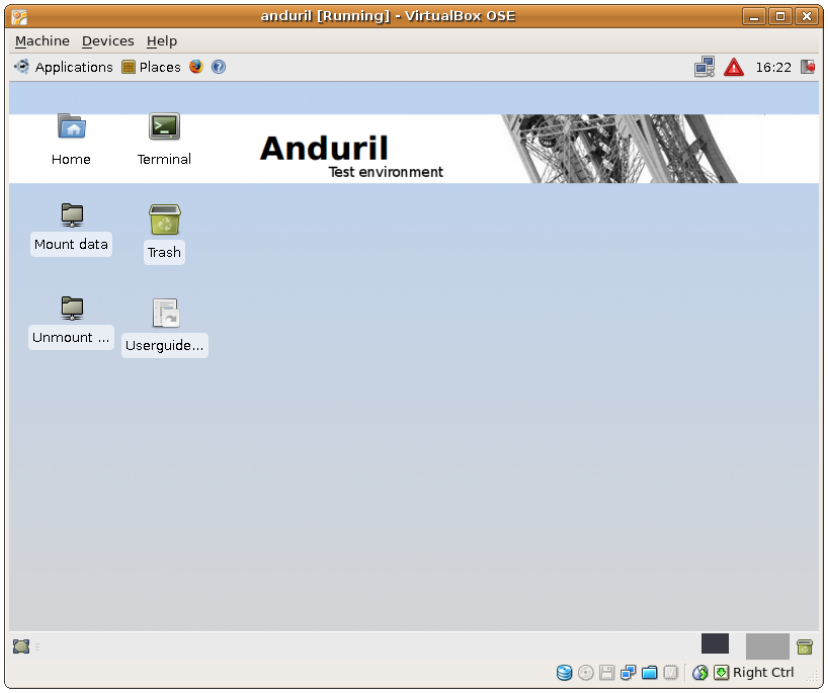


Figure 6: Virtual Linux desktop.

2.2 Debian package installation

This example works on Ubuntu Maverick. For other versions use the matching name in the repository lists.

Add the following repositories to your 3rd party sources, by adding them in the file `/etc/apt/sources.list.d/anduril.list` or using the update manager:

```
deb http://cran.at.r-project.org/bin/linux/ubuntu precise/
deb http://www.anduril.org/anduril/linux/ binary/
```

Add the GPG signature key with the commands:

```
wget http://www.anduril.org/anduril/linux/anduril_pub.gpg -O anduril.key
sudo apt-key add anduril.key
```

Update your package lists (`sudo apt-get update`), and install anduril (`sudo apt-get install anduril`). Note that R packages must be installed separately. A lot of the components use Bioconductor packages not included in the Ubuntu repository.

To manually install Anduril in a Debian/Ubuntu system, download the Debian package from

```
http://www.anduril.org/anduril
```

and type `gdebi [Name of the downloaded file]`. This program wraps the Anduril binary package and describes library dependencies, which are solved by `gdebi` (in order to install `gdebi` use `sudo apt-get install gdebi`). You still need to add the repositories for R and canonical like shown earlier. Once installed, a man page shows the command line options; type

```
man anduril
```

The environment variable `ANDURIL_HOME` must be set to the Anduril installation to be used, especially in case several versions are available in the system.

2.3 Binary package installation

The binary package contains the Anduril system and Java libraries needed by the Anduril core and some components, but other dependencies such as Java and R need to be installed. Java source has been precompiled to JAR and class files.

First, install external dependencies:

- Java SE JDK 1.6 or greater: `http://java.sun.com/`

- R is needed by many components; a recent version is recommended: http://www.r-project.org/
- Bioconductor for R is likewise needed by many components. Install a basic version and add other packages as needed. http://www.bioconductor.org
- LaTeX for documentation generating components (see Section 4.2.9). On Windows, it is easiest to install MiKTeX: http://miktex.org/
- Graphviz for visualizing networks, such as workflows: http://www.graphviz.org/.

Then, follow these steps:

1. Download the installation ZIP file and uncompress it in the installation directory. See the `readme.html` file for package contents.
2. In `bin/anduril` (or `bin/anduril.bat` on Windows) you can find the Command Line Interface script for Anduril. Either copy it to some directory in your PATH or add the `bin` directory to your PATH.
3. Set the environment variable `ANDURIL_HOME` to the absolute home directory of the Anduril project. This is the directory that contains `anduril.jar`. On Windows, use forward slashes (/) instead of backslashes (\). Alternatively, you can edit the `anduril` script and hard-code a default home directory by editing the variable `DEFAULT_ANDURIL_HOME`. See the script file for details. Verify that the script is working by executing `anduril` on the command line. You should see a help message.
4. If you need components written in R, install the R support package `componentSkeleton`. On Windows, install the binary package `r/componentSkeleton_X.Y.zip`, where X.Y is a version number. On Unix, install the source package `r/componentSkeleton_X.Y.tar.gz`. You can install R packages either with administrator priviledges (recommended) or as a regular user. For the latter case, you may need to define the `R_LIBS_USER` environment variable that points to your R package repository.
5. If you want to automatically install requirements of components, such as R packages, use the InstallRequirements component:
   `anduril run-component InstallRequirements`
   Refer to the documentation of InstallRequirements prior to running it.
6. The Unix/Cygwin command `bin/anduril-runtest` can be used to execute tests for the Anduril engine.
7. Test cases for microarray components can be executed with `bin/anduril-runtest microarray`. This may take a while. Some components may fail if, for example, their external requirements are not satisfied. Make

sure that the components you need are working. For more information on component testing, see Section 3.2.3.

3 Using Anduril

Anduril can be used in two ways: from Eclipse, a popular multipurpose GUI; or from the command line. Both methods enable local or remote workflow execution.

3.1 Eclipse interface

Eclipse (http://www.eclipse.org/) can be used to construct and execute Anduril workflows. An Eclipse plugin, AndurilEclipse, provides Anduril support for Eclipse. The plugin provides (1) an AndurilScript editor that is used to construct Anduril workflows and (2) facilities the execution of workflows by invoking the Anduril workflow engine. See Section 5 for the syntax of AndurilScript. The editor supports syntax highlighting, error markers that show locations of errors, and automatic completion of component names. The text editor is shown in Figure 7.

3.1.1 Eclipse plugin installation

The AndurilEclipse plugin is available on the Eclipse plugin site at

`http://csbi.ltdk.helsinki.fi/anduril/eclipse`

Installation is done using the Software Updates feature in Eclipse rather than downloading manually. The plugin has been tested on Eclipse 3.2–4.2. The following instructions

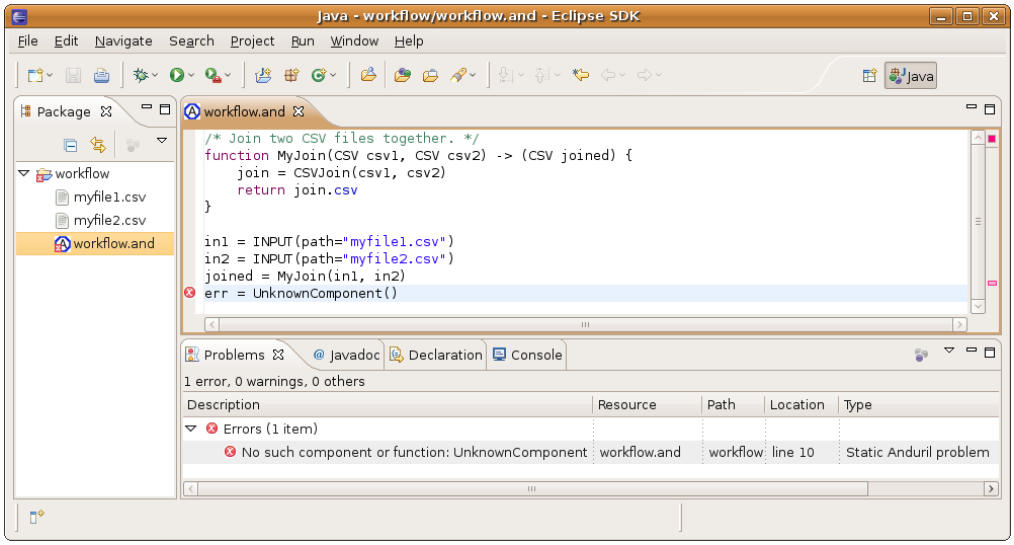


Figure 7: Using AndurilEclipse to construct workflows.

are for Eclipse 3.4, but they can be applied to other Eclipse versions as well.

1. Download and install Eclipse from http://www.eclipse.org/.
2. In Eclipse, select ”Help” → ”Software Updates...”
3. In ”Software Updates and Add-ons” → ”Available Software”, add the Anduril-Eclipse plugin site by selecting ”Add Site...” and entering `http://csbi.ltdk.helsinki.fi/anduril/eclipse` as location (Figure 8, 1–2).
4. Select the AndurilEclipse plugin and press ”Install’ (Figure 8, 3).
5. After being prompted for license agreement the plugin should be installed. You may also have to restart Eclipse.

3.1.2 Configuring the plugin

After installation, it may be necessary to configure AndurilEclipse by entering the installation path of Anduril and adding additional bundles. Configuration is done from ”Window” → ”Preferences” → ”Anduril Preferences” (see Figure 9).

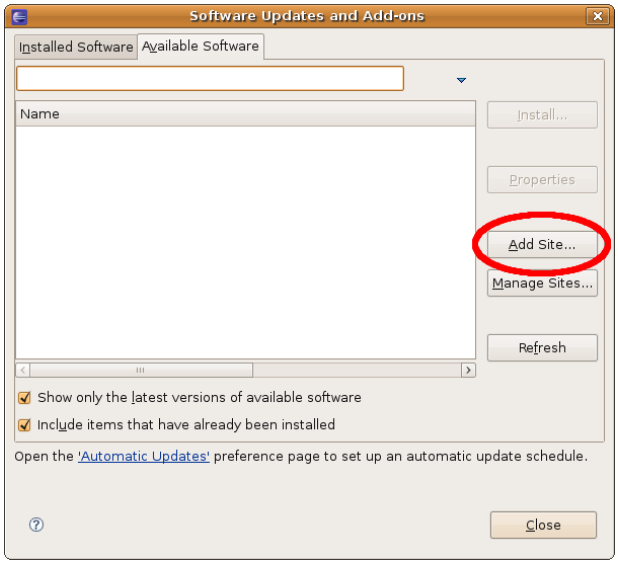
If the environment variable `ANDURIL_HOME` is set, AndurilEclipse uses that variable as the default installation location. Otherwise, the location must be set manually. The location is the directory containing `anduril.jar`.

If additional resource bundles not present in Anduril distribution are needed, their paths should be entered in Anduril Preferences. The path of a bundle is a directory that contains `bundle.xml` (see Section 8 for details).

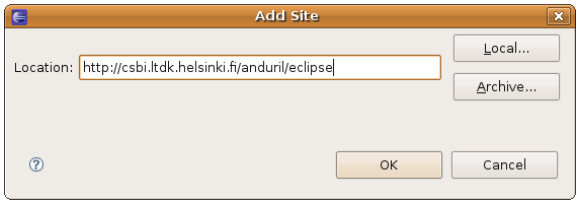
3.1.3 Constructing workflows

In Eclipse, all files such as AndurilScript programs are part of a project, so first you need to create a project (”New” → ”New project”). You can select a general project type such as ”Project → ”General”. To create a new workflow, select ”New” → ”Other” → ”Anduril workflow”. The file extension of AndurilScript files should be ”`and`”. You can now edit the AndurilScript code using the text editor provided by Eclipse. To automatically complete component names, use the control–space key combination.

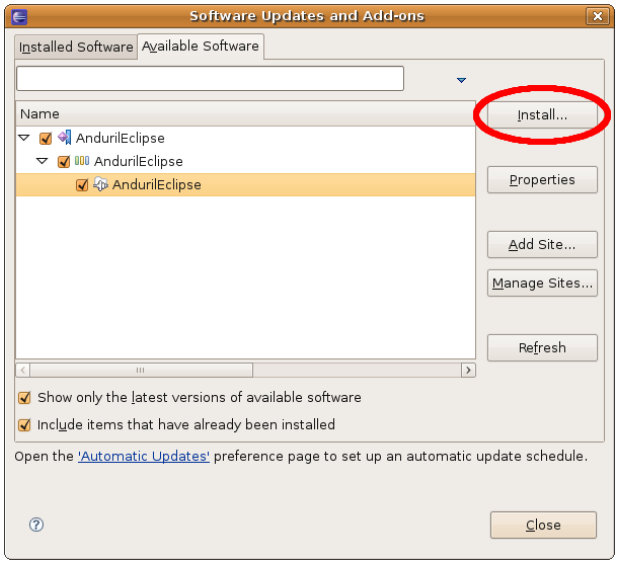
The AndurilScript source file is parsed continuously while editing and any errors are highlighted in the editor. Detailed error messages are available in the Problems tab. If the Problems tab is not visible, select ”Window” → ”Show view” → ”Problems”.



1



2



3

Figure 8: Eclipse plugin installation.

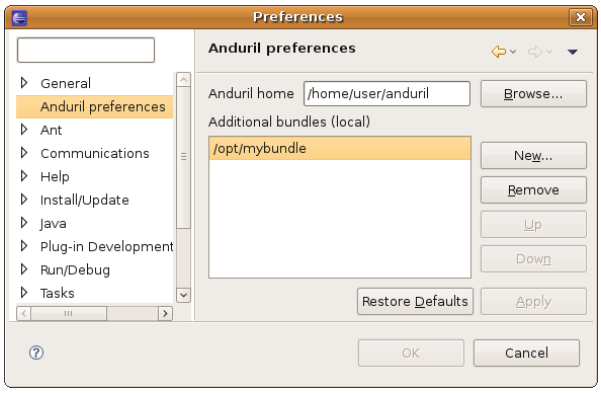


Figure 9: Configuring the AndurilEclipse plugin.

3.1.4 Executing a workflow

The workflow execution engine is invoked using the run facilities of Eclipse. First, create an execution setup using ”Run” → ”Run configurations” → ”Anduril workflow” (see Figure 10). Each workflow should have its own entry in ”Run configurations”. The path to the workflow configuration is given with ”Configuration file” and is filled automatically. ”Data directory” is the location of data files that are referred to using relative instead of absolute file names. ”Execution directory” is the directory where results are written.

The workflow can be executed from the execution configuration panel using ”Run”. Also, recently executed configurations are available in ”Run” → ”Run history” in the text editor. Components print status messages to the console. If the console is not visible, select ”Window” → ”Show view” → ”Console”. If execution of any component instance fails, the corresponding source line is highlighted with an error marker. These dynamic errors are cleared at the start of the next execution.

Normally, Anduril only executes component instances whose configuration has been changed after the previous run or whose execution failed on the previous run. This can be overridden using the ”Force components” field that contains a comma-separated list of component instance names that are forced to execute. For example, `c1,c2` causes components instances c1 and c2 to be executed. The special value ∗ re-executes all component instances.

The maximum number of concurrent component execution threads is controlled with ”Number of threads”.

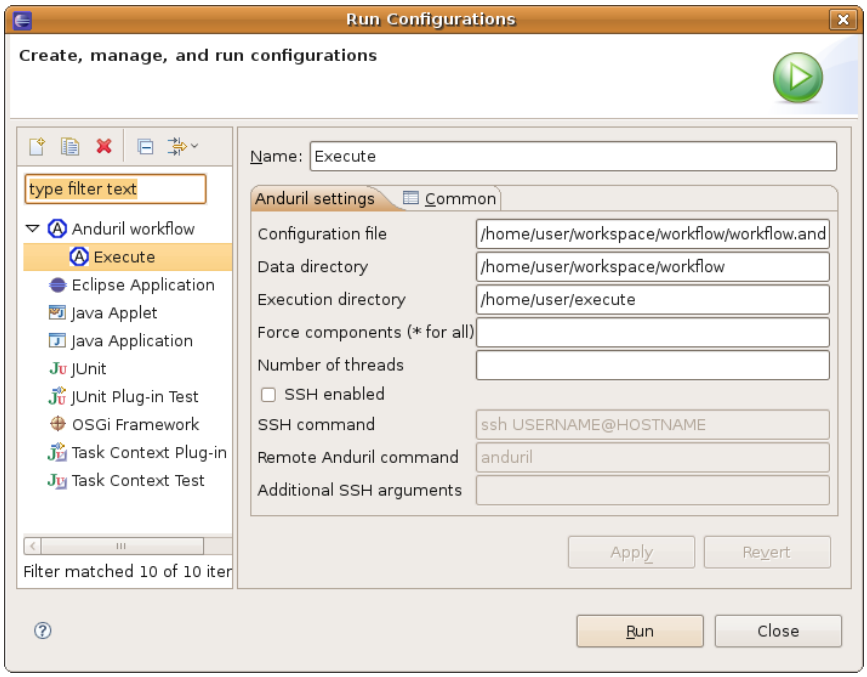


Figure 10: Creating a workflow execution setup.

3.1.5 Remote execution over SSH

The workflow engine can be invoked either locally (default) or remotely over SSH. In remote invocation, both the components and the engine are executed on the remote machine and Anduril must be installed on the remote machine. Interpretation of file paths is different in remote invocation. The configuration file points to the local machine, but other paths (data directory and execution directory) are paths on the remote server. All data files must be present on the remote server. Additional AndurilScript source files, included into the main source using `include` statements (see Section 5.5), must be present on the remote server.

To enable SSH invocation, select ”SSH enabled” in ”Run Configurations” and edit the SSH command by providing the host name and the remote user name. You may also choose to use a different SSH client than the default (`ssh`). On Windows, `plink` from the PuTTY toolkit (`http://www.chiark.greenend.org.uk/~sgtatham/putty/`) can be used. AndurilEclipse invokes the `anduril` command line utility on the remote server; the name of the utility can be customized using ”Remote Anduril command”. Workflow execution can be further customized using ”Additional SSH arguments” that are passed to `anduril`. For example, to import additional bundles, use `-b BUNDLE`. See

Section 3.2 for details.

AndurilEclipse does not cache SSH passwords. It is strongly recommended to automate SSH authentication using public key tokens and an SSH agent program running on the local machine. Using these techniques, no user interaction is necessary for authentication. For pointers on public key authentication, see, e.g., http://sial.org/howto/openssh/publickey-auth/ and http://unixwiz.net/techtips/ssh-agent-forwarding.html. For PuTTY, an agent is available as `Pageant`. If user interaction is needed for authentication, ensure that Eclipse has an input console available.

3.2 Command line interface

The command line tool is called `anduril`. The tool implements several functionalities such as running workflows, running individual components, generating manual pages, etc. The functionality is determined by a command, which is the first argument for `anduril`. All commands are listed in Table 2. For example, "`anduril run`" executes a workflow and "`anduril run-component`" executes a single component. Some commands (e.g., executing workflow engine unit tests) are mainly for developers, while others are for end users. The end user commands are elaborated in the following sections.

Commands have additional arguments and options, some of which may be mandatory. Some options are shared by all commands, while some are specific to a command. All options are listed in Table 3.

Running "`anduril --help`" shows the list of commands and "`anduril <command> --help`" shows the list of options for the given command.

3.2.1 Executing a workflow

A workflow is executed with

```
anduril run WORKFLOW-FILE -d EXECUTION-DIRECTORY
```

Here, `WORKFLOW-FILE` is a file containing workflow configuration and `EXECUTION-DIRECTORY` is the execution directory where component outputs are written. If `-d EXECUTION-DIRECTORY` is not given, the execution directory defaults to current directory. If `WORKFLOW-FILE` is "`-`" (without quotes), workflow configuration is read from standard input.

Command	Usage	Description
build-doc	`destination-dir`	Build component interface and data type HTML docs.
clean	`<workflow-file | "-"> -d EXECDIR`	Remove obsolete files and directories from an execution directory. These may have been produced by component instances that have since been removed or renamed. Only directories corresponding to current component instances are preserved. Note that the -d argument is mandatory.
list	`<workflow-file | "-"> [name-patterns]`	Print a summary of output file contents of one or more components. If no further arguments are given, ports of all component instances are printed. If arguments are given, they are Java regular expressions that are matched against component instance names. Hierarchical components are separated with '-'.
run	`<workflow-file | "-">`	Execute a workflow. Workflow configuration is read from <workflow-file> or, if <workflow-file> is "-", from standard input. The state of previous workflow execution is read from disk and only changed and failed components are executed. This can be overriden with –force and –force-all.
run-component	`<component or function name> [-I ...] [-O ...] [-P ...]`	Execute a single component. Input file paths (-I) must be given for mandatory input ports. Parameter values (-P) must be given for parameters with no default.
test	`[components...] [-B bundle] -b bundle`	Run test cases for components. Names of components are given as parameters, or alternatively, all components that have test cases are tested. The set of enabled test cases can be fine tuned with –test-cases.
test-networks	`[workflow-names] -b bundle1 [-b bundle2]`	Run workflow tests for one or more bundles. Each test is defined by a workflow configuration and an expected output directory. [workflow-names] gives names of directories. If given, only tests located in named directories are executed.
unittest	`[unit-test class names] -b techtest -d DIR`	Execute unit tests for the engine.

Table 2: List of CLI commands. In usage, `[...]` denotes an optional argument, `<...>` denotes a mandatory argument and `x|y` denotes alternatives.

Option	Commands	Description
`-a arg`	all	Text file containing additional arguments (there can be several -a arguments)
`--auto-bundles`	all	Automatically import build-in bundles (where autoload=true in bundle.xml).This is the default for commands run and run-component.
`-B arg`	build-doc, test, test-networks	Resource bundle root directory (there can be several -b arguments). If -B is given, the corresponding -c and -t arguments are not needed.
`-b arg`	build-doc, clean, list, run, run-component, test, test-networks, unittest	Target bundle root directory (there can be several -b arguments). If -b is given, the corresponding -c and -t arguments are not needed.
`-c arg`	build-doc, clean, list, run, run-component, test	Component repository root directory (there can be several -c arguments)
`-d arg`	clean, list, run, run-component, test, test-networks, unittest	Network execution directory (default: execute)
`--data-dir arg`	run	Location of data files for input components. If not given, the directory containing workflow configuration is used. If <workflow-file> is "-", this directory is also searched for included workflow files.
`--dry`	clean	Dry run: show unused directories but do not delete anything
`--dry`	run	Generate a network for the pipeline and create a state based on the network, but do not execute the pipeline.
`--exclude arg`	unittest	List of class names whose tests are excluded
`--exec-mode arg`	all	Specifies a mode of execution. Possible values are local, remote, slurm and prefix. Default value is local.
`--fast`	unittest	Only execute a subset of tests that run fast
`--force arg`	run	Force execution of the following components (in comma-separated list), even if their configuration is unchanged
`--force-all`	run	Force execution of all components, even if their configuration is unchanged
`--help`	all	Print help
`--hosts arg`	run	Remote host configuration file.
`-I arg`	run-component	Input path for a port, with format -I portname=filename
`--interface arg`	all	Type of user interface (plain or curses). Default: plain
`--java-heap arg`	all	Heap size for Java components in MB (default: 200)
`-L`	run-component	List in- and out-ports, and parameters for component or function
`--log arg`	all	Directory for writing log files (default: log). Anduril assumes ownership of this directory and may delete any files from it.
`--min-space`	run	Minimize disk storage by removing unused intermediate output.
`--no-auto-bundles`	clean, run	Do not automatically import build-in bundles (the opposite of –auto-bundles)
`-O arg`	run-component	Output path for a port, with format -O portname=filename
`-P arg`	run-component	Value of parameter, with format -P paramname=value
`--perl-exec arg`	all	Perl execution command (default: perl)
`--prefix arg`	all	Specifies a prefix command to be executed when running a component. Requires the –exec-mode prefix option.
`--print-latex-usage`	default	Print usage as LaTeX table format (for Anduril maintenance)
`--python-exec arg`	all	Python execution command (default: python)
`--R-exec arg`	all	R execution command (default: R)
`--retain-network`	run	Do not write state file at the end, only when components finish, in order to avoid overwriting the whole network state at once.
`-t arg`	build-doc, clean, list, run, run-component, test	Data type XML file (there can be several -t arguments)
`--test-cases arg`	test	Execute only these test cases. Comma-separated list of test case names, e.g., 'case1,case2'. If omitted, all test cases of selected components are executed.

You may also need to import components and data types that the workflow needs. Components and data types bundled with the Anduril distribution are found automatically and you do not need to give additional parameters. Automatic importing of built-in components can be disabled with `--no-auto-bundles`. Other bundles can be imported using using `-b BUNDLE-DIRECTORY`. Components and data type definitions may also be located outside a bundle directory structure; these can be imported using `-c COMPONENT-REPOSITORY` and `-t DATA-TYPE-FILE`. Here, `COMPONENT-REPOSITORY` refers to a directory that contains component directories; it is not a component directory itself. See Section 8 for the file structure of bundles, component repositories and data type files.

There are various options that control workflow execution. The maximum number of concurrent component execution threads is controlled with `--threads NUM`. For Java components, maximum heap size in megabytes is given with `--java-heap MEMORY`.

After the network is executed, final results (produced with `OUTPUT` components) are stored in the directory `output` under the execution directory. Intermediate results of components are stored in subdirectories named after the component instances.

Example The workflow configuration is in `./workflow.and` and it uses additional components from the bundle in `/opt/mybundle`. Component outputs are stored in `./execute`. The c1 and c2 component instances are forced to be executed, even if they were succesfully executed on the previous run. The number of component threads is restricted to three and Java components are given a maximum of 1 GB (1024 MB) heap space. The command (all on one line) to execute the workflow is

```
anduril run ./workflow.and -d ./execute -b /opt/mybundle
--force c1,c2 --threads 3 --java-heap 1024
```

3.2.2 Executing a single component

Occasionally it is useful to execute a single Anduril component without placing it on any workflow. This is done with

```
anduril run-component COMPONENT-NAME -I port=FILE
-P name=VALUE -O port=FILE -d EXECUTION-DIRECTORY
```

Here, `COMPONENT-NAME` is the name of an Anduril component and `-d EXECUTION-DIRECTORY` is as in previous section. The `-I port=FILE` options give file paths corresponding to input ports of the component so that `port` is the name of the port and

FILE is the path to the file. There can be several -I options, one for each port. Every mandatory input port must have an associated file. Files for output ports are given with -O options; they are optional. Values for simple parameters are given with -P name=VALUE, where name is the name of the parameter and VALUE is the value.

Anduril creates a small temporary workflow containing the named component. The workflow is executed like a regular workflow, with output placed in the directory given by -d. If -O options are given, component outputs are copied to given files.

Example You want to execute the component MyComponent, with input ports in1 and in2, integer parameter p1 and output ports out1 and out2. Input and output ports have the type CSV. Input files corresponding to in1 and in2 are ./in1.csv and in2.csv. The value for parameter p1 is 42. Outputs are placed into files ./out1.csv and ./out2.csv. Execution directory is ./execute. To command (all on one line) to execute the component is

```
anduril run-component MyComponent -I in1=./in1.csv
-I in2=./in2.csv -P p1=42 -O out1=./out1.csv -O out2=./out2.csv
-d ./execute
```

Sometimes, even more useful is to execute a component with the same configuration that was used to launch it in a workflow. This is possible with the _launch script that is written to each component instance's execution folder. This is explained in further detail under 3.6.

3.2.3 Executing component test cases

Component test cases are executed with

```
anduril test [COMPONENT-NAMES] -b BUNDLE [-B BUNDLE]
[--test-cases CASE1,CASE2] -d EXECUTION-DIRECTORY
```

The only mandatory parameter is -b, which specifies the bundle(s) from which components are selected. The -B specifies additional bundles whose components and functions are used for the execution of the target bundle specified in -b. If COMPONENT-NAMES is given, only these components are tested; otherwise, all components from selected bundle(s) are tested. If --test-cases is given, only selected test cases are executed. Test case names must match the ones given on component HTML manual pages, e.g. case1.

The engine produces a report that shows which test cases pass and fail. Outputs of components are available in the execution directory and they can be compared to expected outputs located in the component bundle (see Section 6.1 for details on directory layout).

In some situations, test cases may fail for components that do not have genuine errors. The execution environment may lack third-party libraries required by the component. The environment may have a different version of a library (or R) that produces slightly different results; whether this is an error depends on the situation. Test cases for components that query external databases and other resources may fail when the database is updated; this is generally not an error but requires updating the test case to current output by the component author. The external resource may also be temporarily offline.

Example The following example executes `case1` and `case2` of components MyComp1 and MyComp2. If MyComp2 does not have `case2`, then only `case1` for MyComp2 is executed.

```
anduril test MyComp1 MyComp2 -b myTargetBundle
-B myResourceBundle --test-cases case1,case2 -d ./execute
```

3.2.4 Executing a workflow with the #! runner

The `#!` (hash bang) runner script can be used to execute an Anduril script and to save the command line switches in the workflow file itself. Add the `$ANDURIL_HOME/bin/anduril-runner` script in your path, and add the following line in the beginning of the Anduril workflow file:

```
#!/usr/bin/env anduril-runner
```

Change your workflow file to have execution rights, and run it with `./workflow.and`.

3.2.5 Advanced command line usage and debugging

Sometimes running the workflow can fail. To work around these situations, many strategies can be used.

Normally, Anduril only executes component instances whose configuration has been changed since the previous run or whose execution failed on the previous run. This can be overridden with `--force c1,c2,...`, which causes instances c1, c2, ... to be executed. Re-execution of the whole workflow can be forced with `--force-all`.

For R and Python components, the R and Python executables are given with `--R-exec` and `--python-exec`, respectively.

While editing the workflow, you may want to comment parts of the code. This will result to the network being altered and the state of already executed component instances will be lost when Anduril writes the final state of the network. Sometimes changes in the dynamic network execution may have a similar effect. This could potentially happen if the workflow execution was interrupted abnormally. Therefore some users have picked the habit of using `--retain-network`, which prevents the default behavior. The network state is only written when each component finishes, so already existing component states remain unaffected. Note that to get rid of their files, you would need to use the clean command anyway, so there is nothing lost before you want to clean the execution directory.

Example You can add any of the Anduril command line switches in the workflow file.

```
#!/usr/bin/env anduril-runner
//$ -B myResourceBundle
//$ -d ./execution_folder
```

3.3 Apache Ant interface

Anduril has many things in common with Apache Ant. First of all, Ant is used as a build tool for the development of Anduril itself. Secondly, Ant scripts can be used within Anduril pipelines. They can be nicely executed with a dedicated Ant component or wrapped into a StandardProcess or a BashEvaluate call. Thirdly, AndurilScripts can be invoked as Ant tasks. Anduril provides its own custom task library for these invocations so that you do not necessarily need any command line interface calls (`exec` task). Ant and Anduril are both workflow engines with their own advantages. We try to encourage the use of best alternatives by keeping them interoperable.

3.3.1 Executing a workflow

Anduril workflows can be executed within the Ant scripts, which enables a convenient management of the files and paths. A workflow can be executed with the run task provided by `fi.helsinki.ltdk.csbl.anduril.core.ant.RunTask`. Forced re-execution is best used by overriding the default setting from the command line (`ant run -Dforce=this,that`). The options of this task are shown in Table 4

Attribute	Value	Required	Description
autobundles	boolean, *default:* true	false	Automatically import build-in bundles
execmode	local, remote, slurm or prefix	false	Mode of execution
executionDir	folder, *default:* execute	false	Execution folder
force	a comma separated list of instances or an asterisk for all	false	Forced re-exceution of the given instances
javaHeap	integer of Megabytes, *default:* 200	false	Heap size for Java components
launchercpref	path reference	false	Class path for the Java components
logDir	folder, *default:* log	false	Log folder
prefix	command	false	Prefix command to execute with each component
threads	integer, *default:* 4	false	Number of concurrent component threads
workflow	AndurilScript file	true	The main executable for the workflow

Table 4: List of run task arguments.

Listing 1 shows how to run the already mentioned `workflow.and` in Ant. This example refers to a one external bundle (`/opt/mybundle`) that has not been loaded automatically.

Listing 1: This Apache Ant script that can be used to launch a simple Anduril pipeline.

```
<?xml version="1.0"?>
<project name          = "MyAndurilProject"
         basedir       = ".">

  <property environment="env" />
  <property name="pipeline"      location="workflow.and"/>
  <property name="exec.dir"      location="execute"/>
  <property name="log.dir"       location="${exec.dir}/log"/>
  <property name="myOwn.bundle" location="/opt/mybundle"/>

  <path id="base.path">
   <pathelement location="${env.ANDURIL_HOME}/anduril.jar"/>
   <fileset dir="${env.ANDURIL_HOME}/core/lib">
     <include name="**/*.jar"/>
   </fileset>
  </path>
```

```
17
18   <target name="clean">
19     <taskdef name="anduril-clean" classpathref="base.path"
20       classname="fi.helsinki.ltdk.csbl.anduril.core.ant.CleanTask"
21     />
22     <anduril-clean executionDir="${exec.dir}"
23                              logDir="${log.dir}"
24                           workflow="${pipeline}"
25                      launchercpref="base.path"
26                        autobundles="false">
27       <bundle file="${myOwn.bundle}"/>
28       <sysprop name="PROJECT_NAME" value="${ant.project.name}" />
29     </anduril-clean>
30   </target>
31
32   <target name="run">
33     <taskdef name="anduril-run" classpathref="base.path"
34       classname="fi.helsinki.ltdk.csbl.anduril.core.ant.RunTask"
35     />
36     <property name="force" value=""/>
37     <anduril-run executionDir="${exec.dir}"
38                            logDir="${log.dir}"
39                         workflow="${pipeline}"
40                         javaHeap="200"
41                          threads="4"
42                      autobundles="true"
43                    launchercpref="base.path"
44                            hosts="hosts.cfg"
45                            force="${force}">
46       <bundle file="${myOwn.bundle}"/>
47       <sysprop name="PROJECT_NAME" value="${ant.project.name}" />
48     </anduril-run>
49   </target>
50
51 </project>
```

3.4 Anduril graphical user interface

Anduril *graphical user interface* (GUI) can be generated and launched with Ant tasks, which enables running an Anduril function graphically. The GUI form is generated by `fi.helsinki.ltdk.csbl.anduril.core.ant.FormConstructor`. The GUI workflow is executed with the run task provided by `fi.helsinki.ltdk.csbl.anduril.core.ant.RunComponentTask`.

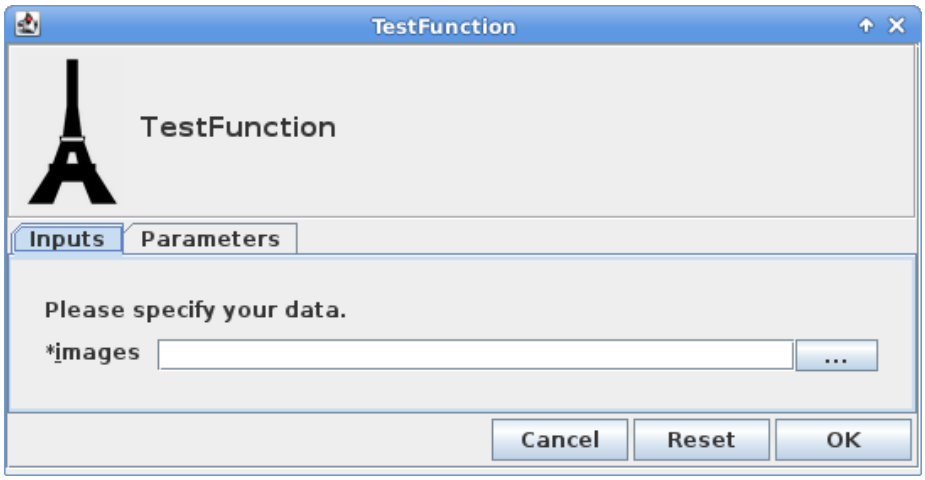


Figure 11: Anduril GUI example.

Anduril GUI generator is started with the `anduril-GUI-generate` command. Enter the exact name of the Anduril public function, for example: "`HistogramPlot`". `FormConstuctor` generates the given function GUI files (`build.xml` and `build.properties`) in to the `[FunctionName]GUI` folder.

Anduril GUI is started from the command line by typing `ant` in the `[FunctionName]GUI` folder. Specify your inputs and parameters in to the GUI then press OK. The function is executed in the `exec` folder. Log files are stored in the same folder. The outputs defined by the function are compressed in a ZIP archive named `results.zip`.

Error	Description
"Unknown function"	Check name of the function
Can't press OK	Check required fields ("*")
"Anduril failed!"	Check inputs and parameters
"Nothing to execute"	Function already executed, no inputs or parameters changed

Table 5: List of errors.

3.5 Browsing the execution folder

As the execution folder fills with instance subfolders, it may get difficult to navigate between the inputs and output files of a component instance. To help the browsing of the execution folder, there is a Python based console tool called `anduril-result-browser`.

Figure 12: Anduril result browser window example.

Type `$ANDURIL_HOME/bin/anduril-result-browser -h` to learn about the switches to the program.

- Green items are input or output ports. They are links to files or folders. By clicking them, an appropriate viewer is launched. You can change the launchers in the configuration file (defaults to `/.config/anduril/resultbrowser`).
- Blue items are other instances that are connected to the currently viewed one. Clicking them will change the view to that instance.
- Back and forward buttons (f,b keys) are used to navigate.
- Reload button (r) reloads the state file.
- x and e keys exit.

Here is an example of an alternate configuration file for the browser:

```
folder = geeqie  %f &
array = xterm -fn 10x20 -e anduril-result-browser -a %f
.csv = xterm -fn 10x20 -e ncsv %f &
.zip = squeeze %f &
none = xterm -fn 10x20 -e ncsv %f &
default = less %f
```

3.6 Isolated execution of a component instance in a workflow for debugging

A workflow contains many parallel instances and so it is possible to get dozens of replicas of a single error, or sometimes only one instance of many otherwise identical parallel instances could fail due to a bug that only occurs with certain inputs and parameters. Even though Anduril writes log files for everything, it is sometimes easier to run just a single failing step, to get to the root of the issue.

When a component instance is scheduled for execution, Anduril writes a command file in the component's execution folder. The command file contains the inputs, outputs and parameters i.e. everything needed to know about how the component should be executed in the context of the workflow.

Hence it is possible to write a script that emulates the behavior of Anduril, while allowing direct access to the underlying component implementation, which is usually a command line call to a component skeleton implented in the various target languages such as R, Python or Bash. Such a script allows invoking a debugger or any other tools you may wish to use to inspect the problem at hand, with minimal overhead. The script allows you to run the component locally instead of a cluster node, when that is required for debugging, by removing the prefix that Anduril added to the component instance invocation.

The script is automatically written by Anduril itself next to the command file, and is called `_launch`. It can be invoked directly but allows changing any input or output locations and parameters passed to the component.

The actual script, which is used to run the `_launch` script itself is called `anduril-run-instance` and it resides under the `utils` subdirectory of the Anduril source tree. You should thus have that configured on your binary search path. To get documentation and usage examples, either invoke that script directly, or specify the `-h` flag to the `_launch` script. The most useful flag is probably `-d` which redirects all outputs to the folder of your choice, thus lifting the requirement to have write access to the execution directory and avoids perturbing the workflow state.

Especially useful is the ability to let someone else, e.g. a component developer, run only a specific step of the pipeline, without even having write access to the execution directory. All they need is read access to the component's directory and the input files.

The most up-to-date and extensive documentation can be found at the Anduril discussion forum.

3.7 Modes of execution

Anduril supports several modes of executing individual components. The default mode is local, in which all the components are executed on the machine, where Anduril is run. Remote execution mode makes use of SSH and rsync to execute components on remote machines specified in the hosts file. For more details, see Section 5.12. Slurm execution mode makes use of Slurm Workload Manager (`http://slurm.schedmd.com/`) to execute individual components. Finally, prefix execution mode allows an arbitrary command to be executed with each component. This mode is useful for integrating Anduril with any scheduler, as long as it supports launching jobs via command line. Execution mode is specified using `--exec-mode` command line parameter. Possible values are `local` (default), `remote`, `slurm` and `prefix`.

Note, that the component instance annotation `@host="local"`, sets the execution as local, even when `--exec-mode` is set for something else. More about annotations in Section 5.3.

3.7.1 Slurm

In this mode of execution each individual component is submitted to the scheduler, which in turn makes a decision when and when the component is executed. All data is assumed to be stored on a shared storage available to all the nodes participating in the cluster. Also configuration of all the nodes must be identical in terms of installed software used by individual components. Anduril must be run on one of Slurm's nodes (Anduril fails if `srun` is not found). Anduril with Slurm, run the following command.

```
anduril run workflow.and --exec-mode slurm
```

Anduril uses Slurm's srun command to execute components. To provide additional options to srun, use –slurm-args argument. The argument string must be enclosed in double quotes and dashes in the arguments string must be replaced with

```
anduril run workflow.and --exec-mode slurm --slurm-args "%v"
```

If you wish to specify fine-grained resource allocation for a particular component instance, consider using @cpu and @memory annotations. Consult Table 9 more details.

3.7.2 Prefix

In prefix execution mode, an arbitrary command supplied by user is executed as a prefix to each component. A typical use case is making use of a scheduler not supported by Anduril out of the box. For example, to run Anduril with TORQUE Resource Manager (http://www.adaptivecomputing.com/products/open-source/torque/), use the following command.

```
anduril run workflow.and --exec-mode prefix --prefix qsub
```

You can also provide additional arguments to the prefix command. To do that put the whole prefix string in double quotes dashes in the arguments string must be replaced with

```
anduril run workflow.and --exec-mode prefix --prefix "qsub %%arg"
```

You can create a custom prefix command to suit your needs. See the template in the Anduril folder `doc/prefix_template.sh`.

4 Life sciences analysis

In this section, components and data types related to life sciences analysis are introduced. Further information is available in the HTML documentation of the components and data types. The test cases of components can be used as examples of using them; test case files are shown on the component documentation pages. Data type documentation describes the purpose and physical layout of the type and shows which components read and write the data type.

4.1 Data types and file formats

Data types for life sciences analysis are mostly based on CSV (comma separated) files, which store relational-like data. CSV files contain named columns that may hold numeric or string data. The cell separator character in Anduril CSV files is tabulator. Double quotes (") may be used around cell values and they are removed before further processing. Missing values are given as NA. Column names must be present on the first row of the CSV file. In addition, there are general data types (TextFile, BinaryFile and BinaryFolder) that are parent types of the more specialized types.

Numeric data are stored using Matrix, a subtype of CSV, which is a numeric rectangular matrix with named rows and columns. A subtype of Matrix, LogMatrix, is used for logarithm-transformed data (in base 2), such as log-ratios in expression studies.

File formats for more specialized purposes include the following. For working with sequence-level data, the types FASTA, DNARegion and MotifSet are available. Graphs (networks) are stored in XML based GraphML format. Molecular models are stored using SBML. For creating final reports, the reporting subsystem uses the types Latex, PDF, Excel and HTML.

Matrices often hold numeric measurements for samples; samples can be combined using operations such as median or ratio to form new samples groups. The type SampleGroupTable is used to express relationships between samples. Each sample group is defined by group ID, a list of source groups or members, and optionally a group transformation type (ratio, median or mean) that defines how numeric values are derived. One component using SampleGroupTable is SampleCombiner, which computes numeric values for combined samples.

A high-throughput analysis typically produces lists of relevant genes or proteins that are represented using unique identifiers. The SetList type, derived from CSV, can be used to store sets of IDs. Each file contains n ID sets, where n is the number of rows. For simpler experimental settings, the IDList type can store one ID list. Notice that

SampleGroupTable is a subtype of SetList since it is also a collection of (sample) sets. SetList files can be transformed using SetTransformer with operations such as union, intersection and regular expression filtering.

4.2 Components

Component categories include data import; annotation using databases; plots for quality control and result representation; data transformations; report generation; and various analysis methods such as clustering, classification, pathway analysis, GO enrichment, survival analysis, and graph analysis, to name a few.

4.2.1 Basic INPUT and OUTPUT

The INPUT component is used to import data files or directories into the workflow. This component is used in virtually all workflows. INPUT contains a parameter `path` that specifies the path to the file or directory in question. Each distinct file or directory is imported using its own INPUT component invocation. When a workflow is re-executed, INPUT notices if the modification time stamp of the input file has changed and indicates to the engine that component(s) depending on this file need to be re-executed.

URLInput component can be used to fetch data from remote sites using URLs instead of paths. URLInput mimics the behaviour of the INPUT component but it does not recognize modifications automatically. Content of the URL is fetched to the execution folder of the component instance to make it accessible for the down stream components.

Another elementary component is OUTPUT, which is used to direct end results of the analysis into a special `output` directory located in the execution directory. This enables locating primary outputs easily. OUTPUT takes one file or directory as an input.

4.2.2 General purpose processing

Given the central role of the CSV format, there are several general-purpose components for processing CSV files. CSVFilter and CSVJoin can be used to filter rows or join several CSV files together, respectively. CSVTransformer modifies CSV files using R expressions; it is convenient for small tasks such as computing the mean of two matrices. To convert values from one ID space to another (e.g., probe identifiers to gene names), IDConvert is available. IDConvert can also be used to collapse duplicate rows into one, using one of several options for combining numeric and non-numeric columns. ExpandCollapse is used to expand or collapse comma-separated cell values into distinct rows.

For the most flexible CSV operations, TableQuery can be used to process CSV files using SQL. TableQuery takes CSV files `table1`, ..., `tableN` and an SQL query as input. The CSV files are loaded into an in-memory database using HSQLDB, an open source embeddable database engine. A column `col` in `tableN` is referred to as `tableN.col` in the SQL query. The result of the query is written as a CSV file.

For the most general purpose execution, REvaluate is used to execute custom R scripts and StandardProcess is used to execute system commands and other command line applications dealing with files and standard steams. REvaluate enables to include custom R code into a workflow without the need to wrap it in a component. However, if the same R script can be reused in other workflows, it generally should be converted to a component. Finally, the components SearchReplace, FolderCombiner and FolderExtractor process text files and binary folders.

Components specific to numeric matrices include MatrixTranspose, LinearNormalizer and QuantileFilter. The latter two are useful in normalization.

4.2.3 Annotation using databases

The KorvasieniAnnotator component is used to retrieve annotation from Ensembl. Ensembl includes a large variety of annotation, including gene names, genomic locations, Uniprot IDs and Gene Ontology (GO). The list of supported databases is included in KorvasieniAnnotator documentation. The component may use the public Ensembl database or, using a user-specified properties file, a custom database mirror.

BiomartAnnotator provides access to various Biomart-enabled databases, including Ensembl, Reactome, Wormbase, etc. EntrezAnnotator enables queries to Entrez databases, including Entrez Gene and PubMed, using the E-Utilities interface.

The KEGGPathway component retrieves KEGG pathways for Uniprot proteins, or the proteins located on given pathways. Also, KGML2GraphML fetches KEGG pathway topology in XML format.

The PINA component retrieves protein-protein interactions (PPIs) for Uniprot proteins using an integrated database containing several public PPI databases. The interactions may be given in CSV or GraphML format; the latter can be visualized using GraphVisualizer.

More specialized database components include JASPARMotif (transcription factor binding site motifs), EnsemblDNA (fetching DNA sequences from Ensembl), NextGene (finding nearest genes of DNA loci) and RefSNPAnnotator (annotate SNPs).

4.2.4 Statistics, data mining and plotting

A large variety of statistical tests and corrections for multiple hypotheses are implemented in the StatisticalTest component. Also, CorrelationReport is used to compute correlations between numeric variables. KaplanMeier and SNPKaplanMeier (a version tuned for SNP experiments) are used to compute Kaplan-Meier survival estimates.

For data mining, several components provide access to the Weka framework (WekaClusterer, WekaClassifier, WekaTransform and WekaAttributeSelection). Also, ClusterReport implements basic hierarchical clustering and produces output as a dendrogram.

General purpose plotting components include Plot2D and BoxPlot, which use R facilities for high-quality plotting. GraphVisualizer is used to render graphs using one of several layout algorithms available in Graphviz. VennDiagram produces Venn plots that visualize relationships between sets.

4.2.5 Analyzing expression microarrays

For expression microarrays, data are imported using AffyReader (for one-channel Affymetric childs), AgilentReader (two- and one channel Agilent chips) or IlluminaReader (Illumina files). For Agilent, typical preprocessing includes filtering bad quality spots and combining copies of the same probe into one measurement, as well as producing quality control plots. These functions are encapsulated into the AgilentImport function. Typically, replicate samples are combined into one sample group using SampleCombiner. For one-channel chips, fold changes are often computed using the ratio operation. These may be combined so that first two median sample groups are created and their log-ratio is taken.

Differentially expressed genes can be computed using fold change and/or t-test as criteria. Fold changes, or log-ratios, are computed using SampleCombiner. The FoldChange component is a filter that creates ID lists for each sample group based on the log-ratios. The StatisticalTest component is used for statistical testing such as the t-test. Often, the t-test is accompanied with correction for multiple hypotheses using one of the various methods options available in StatisticalTest.

To show the results, GeneTable creates final CSV files that show gene names, descriptions and fold changes for DEGs. DEGReport creates LaTeX sections for a DEG summary and gene name lists.

4.2.6 Analysing single nucleotide polymorphisms

Single nucleotide polymorphisms (SNPs) are genomic alterations where a single base pair varies between alleles of the population. Contemporary sequencing methods are capable of detecting over one million such loci.

SNP data is represented as SNPMatrix tables that consist of rows or samples and columns of markers. Diploid entries of these matrices represent the nucleotide calls of the chromosomes. The construction of these matrices depends on the source of the data. If all samples and their genotypes have been stored into a SNPHelistin database (available in asser.jar), then they can be accessed using the SNPHelistinReader component.

Distributions of genotypes can be calculated using AlleleCounter, which gives the frequencies of `AA` (wild type homozygote), `Aa` (heterozygote), and `aa` (rare homozygote) genotypes and calculates the probability of observing these or more extreme frequencies under the null hypothesis of having a Hardy-Weinberg equilibrium. Distribution of missing values is produced to ease automated quality control.

Genotype frequencies of common markers may be compared between sample sets. GenotypeComparator takes in a case and a control output of AlleleCounter and calculates a risk ratio or odds ratio statistic for them.

4.2.7 Exon array analysis

For the Affymetrix Exon Array platform, data are first imported using AffyReader component, which generates a normalized probeset expression matrix. DiffExon component calculates fold change value for each probeset between two sample groups, maps differentially expressed probesets to their corresponding exons, and finally picks up a list of differentially expressed exons with combined FC from their multiple probesets. After retrieving a list of DEEs, DEGRankFromExon component can rank their corresponding genes based on FC values of DEEs. A list of DEGs from DEGRankFromExon can be imported into GeneToExon component and create a Gene-Exon table which is needed in ExonPlot visualization component. ExonExpression combines the expression data of multiple probesets and outputs a converted exon expression matrix. Exon expression visualization can be done by ExonPlot component. The output figure shows an expression pattern of certain gene, from which we can see the expression differences of all the exons along the sequence of the gene between two sample groups.

4.2.8 Gene Ontology and pathway analysis

After retrieving GO annotation with KorvasieniAnnotator or BiomartAnnotator, enrichment analysis can be done with the GOEnrichment component. This produces a list of GO terms that are statistically enriched in the annotation set. The results of GOEnrichment can be visualized using GraphVisualizer. To find gene products annotated with a specific GO term (or its child term), GOFilter or GOSearch is used. The GOClustering component computes semantic similaries between gene products using their GO annotations and clusters the gene products using hierarchical clustering.

The SigPathway and SPIA components compute statistically affected pathways. The input genes need to be in EntrezGene format; conversion can be done using KorvasieniAnnotator.

4.2.9 Report generation

Creating comprehensive final reports automatically is a central goal of Anduril. The reporting subsystem is based on LaTeX, which is suitable for dynamic generation and produces high quality PDFs. Also, there are components for producing reports in HTML (HTMLReport, SBML2HTML) and Excel format (CSV2Excel).

In the LaTeX framework, individual components produce LaTeX fragments, i.e., incomplete documents. Fragments may refer to auxiliary files such as images. Fragments are combined into a complete LaTeX document by LatexCombiner. Several LatexCombiner instances can be used in a workflow. LatexTemplate provides customized headers and footers and allows to set page margins, for example. Finally, the complete LaTeX document is compiled into PDF using LatexPDF.

CSV files can be formatted in LaTeX format using the CSV2Latex component. For properties and SQL files, the similar components are Properties2Latex and SQL2Latex, respectively.

ConfigurationReport is one of the most important documentation producing components. It produces a topology graph of the component network and a subsection for each component that describes the purpose of the component and values of all simple parameters. ConfigurationReport is an internal component that can access the network data structure that is being executed. If a component implements methods published in a journal, the component description can refer to the publication using BibTeX.

4.3 Worked Examples

This section presents examples on microarray analyses with Anduril. The emphasis here is on the step by step utilization of Anduril to complete the task at hand.

4.3.1 Getting started: random matrices

The first example uses randomly generated matrices to produce PDF and Excel output. The example is artificial, but it has the advantage that it does not require external data files and is easy to understand.

We create two random 50×30 matrices from normal distributions N(0, 1) and N(10, 1). The first columns of the matrices are visualized using a scatter plot. We then compute the arithmetic mean of the two matrices: when A and B are input matrices, the mean is $M[i, j] = (A[i, j] + B[i, j])/2.$ for each position i, j. For each row $i = 1, \ldots, 50$ in the mean matrix, we compute a one-sample t-test that indicates whether the row vector $M[i, \cdot]$ follows normal distribution N(4.9, 1). Note that the true distribution is N(5, 1) so we should get some significant p-values. Finally, the mean matrix and the p-values are exported into an Excel spreadsheet that colors p-values < 0.05 red using conditional formatting.

AndurilScript source code and its network visualization are shown in Figure 13. We wrap matrix creation into a small reusable function. The mean matrix is computed using CSVTransformer, which uses R expressions to create the output matrix. Each transform expression creates a column or a matrix for the output. Since the first column of the Matrix data type must contain row names, we create the names in `transform1`. The scatter plot is instructed to use only the first column of the matrices. In StatisticalTest, the `referenceColumns` parameter is empty since we use a one-sample t-test instead of the more usual two-sample test. Since CSVTransformer produces output as a CSV file, but StatisticalTest expects a Matrix, we need to use the `force` keyword to suppress type checking in this case. DocumentGenerator is a function that takes LaTeX sections as input and produces a compiled PDF that includes workflow configuration in addition to the input sections. The primary outputs (PDF and Excel stylesheet) are copied to the `output` directory using two invocations of the OUTPUT component.

CSV2Excel takes one or more CSV files and places them on their own sheets. In addition, it accepts an optional ”style sheet” file that defines formatting options of cells. The style information is placed in a CSV file (here, named `style.csv`, but the name is arbitrary) and loaded into the workflow using the INPUT component. Here, we bold the header row of each sheet and use conditional formatting on the second sheet containing

```
function RandomMatrix(float mean=0, int columns=30, int rows=50)
  -> (Matrix matrix) {
    matrix = Randomizer(columns=columns, rows=rows, mean=mean,
        distribution="normal")
    return matrix.matrix
}
matrixA = RandomMatrix()        // N(0, 1)
matrixB = RandomMatrix(mean=10) // N(10, 1)
mean = CSVTransformer(matrixA, matrixB,
    transform1="sprintf('MeanRow%02d', 1:nrow(matrix1))",
    transform2="(matrix1+matrix2)/2")
scatter = Plot2D(x=matrixA, y=matrixB,
    xColumns="Column1", yColumns="Column1")
report = DocumentGenerator(scatter.plot)
OUTPUT(report.document)

normality = StatisticalTest(force mean.transformed,
    test="t-test", mean=4.9, // Note: true mean is (10+0)/2
    targetColumns="*", referenceColumns="")
excelStyle = INPUT(path="style.csv")
normalityExcel = CSV2Excel(mean.transformed, normality.pvalues,
    style=excelStyle)
OUTPUT(normalityExcel.excelFile)
```

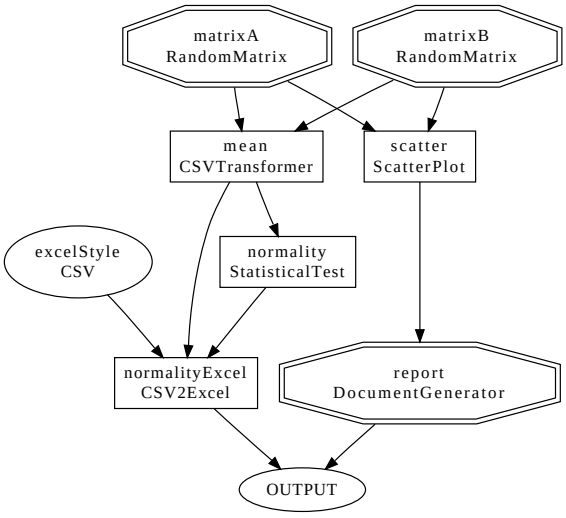


Figure 13: *Top*: Example AndurilScript workflow that processes random matrices. *Bottom*: Main level of the network for the workflow, generated by ConfigurationReport (in DocumentGenerator). Double border nodes represent composite components.

p-values. The contents of the style file is as follows; notice that the values must be separated by tabs.

```
Sheet Row Column  Bold  Condition BGColor
*     1   *       true  NA        NA
2     *   PValue  NA    < 0.05    #ffaaaa
```

4.3.2 Two-channel Agilent arrayCGH microarrays

In this example we give a step-by-step start-up guide on using Anduril for analysis of microarrays. We use a two-channel Agilent ArrayCGH microarrays as example input. The component API can be found from the Anduril website and the reader is encouraged to refer to the documentation for a detailed description of the inputs, outputs and parameters of every component.

Step 1: Input First let us start by defining your inputs and reading in the data. For this we use the ”INPUT” component that simply attaches physical filenames to variables and the ”AgilentReader” component that reads the data in. The locations of the input files are always relative to the location of the script.

```
samples = INPUT(path="samplenames.csv")
datadir = INPUT(path="inputfiles/")
microarrayData = AgilentReader(
    sampleNames     = samples,
    agilent         = datadir,
    filter          = "ControlType!=0 ||
                       (gIsSaturated==1 \&\& rIsSaturated==1)",
    channelColumns  = "gProcessedSignal,gMedianSignal,
                        rProcessedSignal,rMedianSignal",
    probeAnnotation = "SystematicName,Row,Col")
```

Here we have read in data from the directory defined in datadir variable. The samples variable is a text file in the format defined by the ”AgilentReader” component. Since three parameters (”filter”,”channelColumns”,”probeAnnotation”) have their default values, we could have just omitted them.

Step 2: Quality control Once the data has been read in, we want to filter out bad quality probes and normalize our data. The microarrayData variable now contains the outputs of ”AgilentReader”. These include probewise annotations, sample annotations and the intensity matrices for red and green channel, respectively. The intensity matrices

for each channel contain probes on rows and samples as columns. First, let us filter out bad quality probes using the ”QuantileFilter” component.

```
filteredMicroarrayData = QuantileFilter(
    matrix      = microarrayData.green,
    matrix2     = microarrayData.red,
    lowQuantile = 0.05)
```

Anduril does not support reassignment of variable names, so we had to create a new variable filteredMicroarrayData to hold our filtered data. As you can see, we used two fields in the variable microarrayData, namely microarrayData.red and microarrayData.green that hold the probe intensity matrices. Now we can normalize the data with ”ACGHnorm” component and we are also opting to use the default values of the component’s parameters when we don’t explicitly define them.

```
normalizedMicroarrayData = ACGHnorm(
    casechannel = filteredMicroarrayData.matrix,
    control     = filteredMicroarrayData.matrix2)
```

Note that for this step, we have to know which of the two channels was used as case and which as control. Since some of the probes have now been removed from the intensity matrices, we now have to remove them from the probe annotation matrix as well.

```
filteredProbeAnnotation = CSVFilter(
    csv         = microarrayData.probeAnnotation,
    auxiliary   = filteredMicroarrayData.matrix,
    matchColumn = "ProbeName",
    idColumn    = "ProbeName")
```

Generating quality control plots is also very useful. For this end, we use two components this time: ”BoxPlot” and ”Plot2D”.

```
boxPlot      = BoxPlot(normalizedMicroarrayData.casechannel)
scatterPlot  = Plot2D(
    y       = normalizedMicroarrayData.casechannel,
    x       = normalizedMicroarrayData.controlchannel,
    xLabel  = "Fit curve",
    yLabel  = "Residuals",
    title   = "\%s vs. \%s",
    caption = "MA plot for normalized values (\%s vs \%s)")
```

Step 3: Analysis time, and how to write your own functions This time we are only interested in chromosomes 1, 9 and 21. In Array CGH the next step is to segment the intensity matrices. Segmentation is done separately for individual chromosomes, so segmenting a subset of the input data (remember, we have genomewide data) will work just fine. We also might want to rerun our analysis on different chromosomes. Taking a subset of data is easy enough, and in theory we could simply include three consecutive segmentation calls. One Anduril design pattern, however, is the use of functions to circumvent this problem so let's do just that. Function code must preceed their calls in an Anduril script, so we strongly recommend placing them at the top of your script.

First we need to declare the function. In the start of the function declaration

```
function Segmentation(
    CSV normalized,
    AnnotationTable annotation,
    int chromosome) -> (Latex report,
    CSV segments) {
```

we say that as input the function takes two CSV files (”normalized”,”samples”), one AnnotationTable (that is a sub-datatype for CSV) and the chromosome number. We also say that the output of this component is a record that contains one Latex file and one CSV file. As you can see, the curly brackets (”{}”) indicate function boundaries much like they do in most programming langugages such as Java.

Now our normalized data still might have all or most of the chromosomes. We create a filter by only taking those probes from the probe annotation (”annotation”) that contains the input chromosome.

```
chrFilter = CSVFilter(annotation, regexp="Chr="+chromosome)
```

Then, the probe intensities have to be filtered with the annotations that we just filtered.

```
filtNormCaseChan = CSVFilter(csv         = normalized,
                             auxiliary   = chrFilter,
                             matchColumn = "ProbeName")
```

All that is left is running the segmentation on the subset and outputting the results. ”ACGHsegment” takes AnnotationTable in for the input ”geneAnnotation”. CSVFilter, however, results in a CSV file and we just used it to filter our AnnotationTable. Both are CSV files which means that we can ”force” ACGHsegment to take the filtered CSV as an input in place of an AnnotationTable by using the ”force” keyword.

```
segmented  = ACGHsegment(caseChan             = filtNormCaseChan,
                         force geneAnnotation = chrFilter,
                         analysisAlgorithm    = 2)
return record( report    = segmented.report,
               segments  = segmented.segments )
```

Overall, the function looks like this:

```
function Segmentation(CSV normalized,
                      CSV samples,
                      AnnotationTable annotation,
                      int chromosome) -> (Latex report,
                                          CSV segments) {
  chrFilter = CSVFilter(annotation, regexp="Chr="+chromosome)
  filtNormCaseChan = CSVFilter(csv         = normalized,
                               auxiliary   = chrFilter,
                               matchColumn = "ProbeName")
  segmented  = ACGHsegment(caseChan             = filtNormCaseChan,
                           force geneAnnotation = chrFilter,
                           analysisAlgorithm    = 2)
  return record( report    = segmented.report,
                 segments  = segmented.segments )
}
```

Recall that we wanted to segment chromosomes 1, 9 and 21 and we programmed this function to enable us to do just that. We now need to call the function we just created from the main portion of the script. In the function call, all input parameters of the function must be spelled out in their entirity AND in the correct order.

```
chromosome1 = Segmentation(
    normalized = normalizedMicroarrayData.casechannel,
    annotation = filteredProbeAnnotation,
    chromosome = 1)
chromosome9 = Segmentation(
    normalized = normalizedMicroarrayData.casechannel,
    annotation = filteredProbeAnnotation,
    chromosome = 9)
chromosome21 = Segmentation(
    normalized = normalizedMicroarrayData.casechannel,
    annotation = filteredProbeAnnotation,
    chromosome = 21)
```

Step 4: Creating outputs Once we have our analysis ready, the biologist is most certainly anxiously waiting for the results. Since most if not all of our analysis components

(such as ”ACGHsegment”) return their results as LaTeX documents, the easiest way to output all our results is via PDF. Recall that we basically had two different outputs from this one script: first, the analysis results returned by our function ”Segmentation” and the quality control plots created by ”boxPlot” and ”scatterPlot”. What we need to do in order the transform these LaTeX fragments into PDF is to combine them with ”LatexCombiner”, create templates with ”LatexTemplate” and then to transform these into a PDF with ”LatexPDF”. First, let’s do some combining.

```
combinedLatexDocuments = LatexCombiner(
    chromosome1.report,
    chromosome9.report,
    chromosome21.report,
    boxPlot.report,
    scatterPlot.plot,
    sectionTitle = "Our combined latexes")
```

Our Segmentation function returned a field called ”report”, which was assigned to the three variables we created with our function calls. Here we brought them together and added the quality control plots that were created back in step 2. Next, ”LatexTemplate” creates the necessary headers and footers for our PDF, which is build, at last, by combining the template with combinedLatexDocuments.

```
template = LatexTemplate(authors = "Insert your name here",
                         title   = "Name of my analysis")
summaryReport = LatexPDF(combinedLatexDocuments,
                         header  = template.header,
                         footer  = template.footer,
                         useRefs = false)
```

Execution directories in Anduril contain one special directory called ”output”. It is possible to use the OUTPUT component to direct files we want into that directory. This makes it fairly simple to find specific outputs of your Anduril script once it has been executed. This time, we choose to insert into the output folder the PDF and some CSV files that have the results of the segmentation.

```
OUTPUT(summaryReport.document)
OUTPUT(chromosome1.segments)
OUTPUT(chromosome9.segments)
OUTPUT(chromosome21.segments)
```

Although some essential capabilities of Anduril such as annotating your data with information from public databases were not covered here, this example illustrates how to write Anduril scripts that are re-usable, readable and clear.

4.3.3 Integrate fold change information with protein-protein interaction network

In this example we construct an Anduril pipeline for the integration of results from expression array analysis with protein-protein interaction (PPI) network, and the visualization of the network. For this purpose, Anduril provides a component for accessing the PINA database, which integrates PPI-data from six different databases. We filter out the interactions to retain only those in which both proteins are either up or down-regulated.

Step 1: Input First we need annotations for all the genes on the expression array; these annotations are later accessed by their column names. Each gene is annotated as follows:

- gene name (column name : GeneName);
- corresponding Uniprot accession number (Uniprot); and
- fold change (FoldChange).

These annotations are read from precalculated files. Thus, we get an annotation file that includes all the genes on the array:

```
allGenes = INPUT(path="AllGenes.csv")
```

The next input is a list of up and down-regulated genes with their annotations, where the fold change of each gene is greater than 3.

```
FC = INPUT(path="FoldChangeGenes.csv")
```

Step 2: Protein-Protein Interaction (PPI) In this step we fetch PPI from the PINA database for the up and down-regulated genes, annotate the proteins with the data from Step 1 and filter the interactions using fold changes. In the annotation and filtering, we use some additional steps to keep the Anduril pipeline simple and readable, and also to produce intermediate results for further evaluation in other Anduril pipelines. We insert to the PINA component the up and down-regulated genes and the column name that contains the protein identifiers.

```
interactions = PINA(query = FC, column = "Uniprot")
```

PINA returns a list of interaction pairs, and for further processing, we enrich this list with the original annotations. For this purpose we use the TableQuery component,

which processes SQL queries. We define the SQL query as a string that is concatenated from the strings with the + character. As the SQL query includes quotation characters wrapping column names, they need preceding backslash characters, namely for instance

```
table2."queryProtein" = table2.\"queryProtein\".
```

This SQL query combines the annotations with the proteins in the interaction list. From the parameters of the next TableQuery component we can see that table1 includes the allGenes file and table2 includes the list of interactions returned by the PINA component.

```
query =
  """SELECT table2."queryProtein" AS "Uniprot",
         table1."GeneName",
         table1."FoldChange",
         table2."interactingProtein" AS "InteractingProtein",
         (SELECT table1."FoldChange"
            FROM table1
            WHERE table1."Uniprot" = table2."interactingProtein")
             AS "InteractingFoldChange",
         (SELECT table1."GeneName"
            FROM table1
            WHERE table1."Uniprot" = table2."interactingProtein")
             AS "InteractingGeneName",
         (SELECT table1."AffyProbeSet"
            FROM table1
            WHERE table1."Uniprot" = table2."interactingProtein")
             AS "InteractingAffyProbeSet"
            FROM table1, table2
            WHERE table1."Uniprot" = table2."queryProtein" """

interactionsInfo =
    TableQuery(table1 = allGenes, table2 = interactions, query = query)
```

Interactions were fetched only for the up and down-regulated genes with fold change greater than 3. Now, we want to include only those interaction pairs that are regulated toward the same direction. As the interacting protein we found from PINA is probably not from the FoldChangeGenes.csv file, we set a lower fold change threshold for the interacting protein with fold change greater than 1.3. Filtering is also done with the TableQuery component, using a SQL query.

The SQL query is specified as a string inside this Anduril pipeline as follows:

```
query =
  """SELECT table1."*"
```

```
  FROM table1
  WHERE (table1."FoldChange" > 0 AND
         table1."InteractingFoldChange" > 1.3) OR
        (table1."FoldChange" < 0 AND
         table1."InteractingFoldChange" < -1.3) """

FCInteractions = TableQuery(table1 = interactionsInfo, query = query)
```

Step 3: Interaction graph In the third and last step we create a graph of the interaction list using the GraphML representation; then we annotate visualization parameters to the graph and create the visualization.

Now we use the filtered interaction list and get only the protein identifiers for the CSV2GraphML component that creates the GraphML from the PPI. Again, we select the protein identifiers, using the TableQuery component. The resulted table is then inserted to the CSV2GraphML component.

```
query = """SELECT table1."Uniprot", table1."InteractingProtein"
         FROM table1"""

graphList = TableQuery(table1 = FCInteractions, query = query)

graph = CSV2GraphML(matrix = graphList, type = "edgelist",
                    directed = false)
```

The graph annotation has several steps. First we get the template for the annotations using the GraphAnnotator component, which produces a CSV file for annotations. Then we insert new graphical parameters to the annotation CSV file and re-insert it to GraphAnnotator in order to get the graph with new parameters. Finally we visualize the annotated graph.

Getting the graph annotations is achieved with one single call to the GraphAnnotator component, as follows:

```
graphAnnotations = GraphAnnotator(graph = graph)
```

Then we create an SQL query to insert new visualization annotations with the TableQuery component. The annotations inserted here are the title (GeneName) and the color. The color selection is based in the fold change.

```
query =
  """SELECT table1."*",
```

```
    table2."GeneName" AS "title",
    CASE WHEN table2."FoldChange" >= 3
    THEN 'green'
    ELSE (CASE WHEN table2."FoldChange" >= 1.5
          THEN 'blue'
          ELSE (CASE WHEN table2."FoldChange" >= 0
                THEN 'lightblue'
                ELSE (CASE WHEN table2."FoldChange" >= -1.5
                      THEN 'pink'
                      ELSE (CASE WHEN table2."FoldChange" > -3
                            THEN 'purple'
                            ELSE 'red'
                      END )
                END )
          END )
    END AS "color"
  FROM table1, table2
  WHERE table1."name" = table2."Uniprot" """

vertexAnnotation =
     TableQuery(table1 = graphAnnotations.vertexAttributes,
                table2 = allGenes, query = query)
```

The graph annotation and the visualization are done with the GraphAnnotator and the GraphVisualizer components. The final graph visualization is stored in the Anduril output folder.

```
annotatedGraph = GraphAnnotator(graph = graph,
                                vertexAttributes = vertexAnnotation)

interactionGraph = GraphVisualizer(graph = annotatedGraph.graph,
                                   titleAttribute = "title",
                                   layout = "hierarchical")

OUTPUT(annotatedGraph)
```

5 Workflow construction using AndurilScript

Anduril workflows are constructed and configured using a simple yet powerful domain specific script language called AndurilScript. The language is syntactically similar to common programming languages such as Java and R. However, AndurilScript it is much simpler than general purpose programming languages and has been tailored for the purpose of constructing workflows. Both the workflow structure (placement of component instances and their connections) and component parameters are defined using AndurilScript. The syntax of AndurilScript is summarized in Table 6 and the syntactic structures are elaborated in following sections.

Syntax construct	Description	See
`x = MyComponent(in1=y1.port, in2=y2.port, p1=5, p2="q")`	Place an instance of component `MyComponent` onto workflow using name `x`. Connect the port `y1.port` to input port `in1` and `y2.port` to `in2`. Set parameter `p1` to 5 and `p2` to "q".	5.3
`x.out1` `x`	Output port `out1` of component instance `x`. If `x` has only one out-port, port name may be omitted.	5.3
`// Line comment` `/* Multi-line comment */`	Comments.	5.2
`/** Documentation for x */` `x = MyComponent(...)`	Documentation for `x`; must be placed before `x`. Appears in reports.	5.2
`function MyFunc(T1 in1, optional T2 in2, int p1, string p2="x") -> (T3 out1, T4 out2) { /* ... produce x1 and x2 */ return record(out1=x1.out, out2=x2.out) }`	Define a composite component named `MyFunc` that takes two input files and two parameters, and produces two output files. `in2` is optional. `T1`–`T4` are port types. The integer parameter `p1` has no default and the string parameter `p2` has a default. `MyFunc` can be instantiated like a regular component. If there is only one output port, "`return x1.out`" can be used in place of `record`.	5.4
`x = Comp(..., @enabled=false)`	Disable execution of `x` and its dependants.	5.3
`x = Comp(..., @par=rec)`	A mapping from parameter labels onto parameter values; parameters referred in the record instanciate a call to a component. No default.	5.3
`x = Comp(..., @priority=n)`	Set execution priority of `x` relative to default priority n=0. Higher n means higher priority.	5.3
`x = Comp(...)` `y = OtherComp(..., @bind=x)`	Execution of `y` will start only once execution of `x` is successfully completed; no out ports of `x` are requirement as in ports in call to `OtherComp`.	5.3

`x = Comp(..., @execute="always")`
`x = Comp(..., @execute="changed")`
`x = Comp(..., @execute="once")`
	`always`: always execute `x` on every run. `changed`: execute only if configuration changed. `once`: do not re-execute even if changed.	5.3

`include "additional.and"`	Include workflow configuration from file.	5.5

`if X > 8 { /* ... */ }` `} else { /* ... */ }`	Select alternative execution paths using a static Boolean expression; `else` is optional.	5.6

`compareResult = Compare(...)`
`join = switch compareResult {`
`  case a1 = Comp1(...)`
`  case a2 = Comp2(...)`
`  return Join(a1.out, a2.out)`
`}`
	Place a conditional branch on the workflow. The branch component (here, `Compare`) enables one or more of alternatives (here, `a1`, `a2`). The join component ends the branch and output is available under the name `join`. `Comp1` and `Comp2` may be composite components.	5.7

`"String with a \t tab"`	String literal on a single line. Escape character is \.	5.2
`"""Multiline string` `with a \t tab"""`	String literal on multiple lines. Escape character is \.	5.2
`'Raw, uninterpreted string.'`	Raw string literal on a single line.	5.2

`'''Raw multiline string`
`literal with \t etc.`
`uninterpreted'''`
	Raw string literal on multiple lines. Preceding a newline with a \ ignores it.	5.2

`$HOME`	Refer to an environment variable as a string.	5.2
`"String"+X+" concatenation"+5`	String concatenation. Numbers and Booleans can be concatenated to strings.	5.2
`(X == (2+5)*2) && (Y < 2.5)` `Z = 5+2e-1`	Compute a static arithmetic or Boolean expression. Arithmetic operators are `+ - * /`; Boolean operators are `&& || !`; comparison operators are `== != < <= > >=`.	5.2
`true false null`	Literals.	5.2

Table 6: Summary of AndurilScript syntax. Reserved keywords are *emphasized*.

Getting started with AndurilScript is straightforward, as simple workflows can be defined as a sequence of assignment statements. AndurilScript also enables writing complex and flexible workflows by providing support for composite components, conditional processing and spreading workflow configuration into multiple files. Programming-like workflow construction facilitates version history tracking and team work using version control systems such as Subversion or Mercurial.

5.1 Concepts

AndurilScript should be considered as a workflow *construction* rather than a workflow *execution* language, since running an AndurilScript program will create a workflow but not (yet) execute it. The workflow is passed to the workflow execution engine which then executes the workflow. Events occurring during workflow construction are said to be *static* while events during workflow execution are *dynamic*. The distinction is best

Type	Examples	Comments

string	`"abc"`
	`'abc'`
	`"""multiple lines"""`
	`'''multiple lines'''`
		String values allow for concatenations (see Section 5.2).

int, float	`1`
	`-2.5`
	`3.1e-1`
		Numeric values allow for arithmetic and comparison operations (see Section 5.2). When a number is used as the value of an integer parameter, it is rounded to the nearest integer.

boolean	`true` `false`	Boolean values support logic operations (see Section 5.2).
Null	`null`	The `null` constant denotes a missing value. It can be used in the context of optional input ports to mean that the port is not connected.

Record	`x = Comp(...)`
	`y = record(...)`
	`z = {...}`
		`x` is an instance of component `Comp`. Composite components are instantiated in the same way. `y` and `z` (shorthand syntax) are constructed explicitly.

Output port	`x.out1` `x`	When `x` is a component instance, `x.out1` refers to its output port named `out1`. If `x` has only one output port, port name may be omitted.

Table 7: Types available in AndurilScript. Simple types (strings, numbers and Booleans) can be used as values for simple parameters of components. Input ports do not have a type since they are not referred to directly but rather as part of component instantation. Records are collections of name-value pairs. They are created using component instantiation or using explicit constructors.

seen in conditional processing. There are both static (`if` statements, see Section 5.6) and dynamic conditional processing (branch components, see Section 5.7).

Like most programming languages, AndurilScript has variables and values that can be assigned to variables. A variable may refer to a "simple" value, such as a string or a number, but also to a component instance (record) or a port. The available types are listed in Table 7. Note that you cannot assign content produced by components to a variable as component execution takes place dynamically.

Anduril components are functions in AndurilScript. When `C` is a component, "calling" it using `x = C(...)` creates a component instance `x` and places it on the workflow. The parameters `...` specify connections to input ports of `C` and values of simple parameters. An AndurilScript program may define new functions (see Section 5.4), *i.e.*, composite components.

The type system of Anduril is static, strong and uses type inference. This means that the

types of all variables are known statically and type errors are caught before the workflow is executed. In contrast to many other static type systems, type declarations for variables are not needed; rather, Anduril automatically infers the types of variables.

5.2 Basic syntax

An AndurilScript program is a sequence of statements and composite component definitions. Statements are terminated with a newline. Explicit terminating tokens such as ";" are not needed. Statements blocks are separated with "{" and "}".

The most basic statement is the assignment statement, which has the following syntax:

```
name = expression
```

Here, *name* is an identifier composed of letters (a–z or A–Z), digits (0–9) or underscores (_). The first character may not be a digit. On the right hand side, *expression* may evaluate to a simple value (string, number or Boolean), a component instance (see Section 5.3) or an output port (see Section 5.3).

Literals for strings, numbers and Boolean are given with Java-like syntax with the addition of supporting unprocessed raw strings and multiline strings. See Table 7 for examples. Java-like strings are enclosed in double quotes (`"`) and the escape character is `\`. See Table 8 for escape codes. For example, `"\t\"\n"` is a string composed of tab, quote and newline characters. Raw strings are enclosed in single quotes (`'`) and have no escape character.

To create a multiline string containing embedded, literal newlines, use three double quotes (`"""`) or three single quotes (`'''`) to enclose the string instead. For example,

```
"""This is a string spanning

multiple lines"""
```

Raw multiline strings additionally support ignoring literal newlines by using a `\` as the last character of the line. This allows visual line wrapping in the code without actually introducing a newline.

Simple values (Booleans, strings and numbers) can be combined using various comparison and arithmetic operators. The following operators are defined: `&& (and)`, `|| (or)`, `! (not)`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`. Parentheses "(" and ")" are used to modify evaluation order. When applied to strings, the `+` operator is a concatenation operator. It is possible to concatenate strings and other types: for example, `"abc" + 42 + "xyz" == "abc42xyz"` evaluates to true.

You can access environment variables using `$ENVVAR`, where `ENVVAR` is the name of the variable. Environment variables can only be read, not written to.

Code	Description
\n	Line feed
\r	Carriage return
\t	Tabulator
\"	Quote
\\	Backslash

Table 8: Escape codes for string literals.

Records are ordered key-value pair lists that are created automatically in component instantiation, or explicitly using the keyword `record` or short-hand notation `{"key"=value}`. Records support heterogeneous types for both keys and values. Keys are commonly strings or integers, and values may be any types, including other records. The constructor `record` takes key-value pairs as argument but supports only string keys. The short-hand constructor enables using integer keys in addition to strings; it also allows to omit keys (ascending integers are used by default). Records are used in several contexts: to represent component instance outputs (Section 5.3); to return multiple values from a function (Section 5.4); to store any user-defined values; to supplement for-loops (Section 5.9); and to access array data types (Section 5.11). Records are accessed using two syntax forms: `rec.key` for string keys and `rec["key"]` or `rec[1]` for arbitrary keys. The same forms are used to modify existing records: e.g., `rec.key = 5` and `rec[1] = 5`.

Comments are given with Java-like syntax: `//` is a line comment, `/* */` is a multi-line comment, and `/** */` is a documentation string that documents the purpose of a component instance. The documentation string is given on the line preceding the component instance definition.

Examples

```
x = "abc"
y1 = 5.2e-5
y2 = (15 + 4.1) * y1
z = x + "xyz" + y2
cond = (y1 > 0.00001) && !(x == "abc") // This is false

r = record(name1=x, name2="xyz", name3=5.7)
r["name4"] = 10 // same as r.name4 = 10
r[1+2] = r.name4 * 5 // r[3] = 50
r2 = {"name1"=x, "name2"="xyz", "name3"=5.7} // equal to r
r3 = {x, "xyz", 5.7} // keys are 1, 2, 3
```

5.3 Placing component instances on a workflow

Components are instantiated and added to the workflow using an assignment statement with the following format:

```
name = ComponentName(port1=x1, ..., param1=y1, ...)
```

Here, `name` is the name of the component instance and `ComponentName` is the name of the component. Connections to input ports are given with `port1=x1`, where `port1` is a port name and `x1` evaluates to an output port of another component instance that was earlier placed on the workflow (see below). For input ports, the name may be omitted and the short form `x1` can be written; these positional input connections must come before named (long form) connections. Values for simple parameters are given with `param1=y1`, where `param1` is the name of a parameter and `y1` evaluates to a string, number or Boolean, depending on parameter type.

All input port connections must come before simple parameters. All mandatory input ports must be given an incoming connection. Simple parameters that do not have a default value must be given a value. Other port connections and parameters need not be given a value. For optional input ports that are left unconnected, the port name may be omitted from the expression, or the literal `null` may be used as the value.

It is an error to create two component instances with the same name as the name must be unique. Therefore, each name should have only one component instantiation statement that is executed. For if-statements, it is legal to assign to the same name in the if-body and else-body as only one is executed. For composite components, it is legal to have component instances of the same name in distinct sub-workflows as the uniqueness property needs to hold only within a single sub-workflow.

Referencing output ports The result of a component instantiation is a record that contains the output ports of the component as fields. In object oriented terms, a record is a class with public fields only. Output port references are used to connect an output port to an input port of another component instance. If `comp` is a component instance,

```
comp.port
```

refers to the output port named `port` of the component. If the component has only one output port, the port name may be omitted. Note that omitting the name may lead to errors if the component is later modified to include more than one output port.

Bypassing type checking By default, Anduril ensures that port connections do not violate type constraints defined by the types of input and output ports. For example, you cannot connect a PDF port to a CSV port. You can bypass this type checking by using

the `force` keyword before the port connection:

```
name = ComponentName(force port1=x1)
```

Modifying component execution *Annotations* can be used to influence whether a component instance is executed, and whether output files are cached on disk. Component instances can be disabled, forced to execute every time, or prevented from re-executing even if their configuration changes. Also the execution of some components can be prioritized over the execution of others. Annotations are given with the general syntax

```
name = ComponentName(..., @annotation=value)
```

Annotations are special parameters defined for all components. The value of the annotation is stored in the component and used by the execution engine to determine if the instance should be executed. All currently defined annotations are specified in Table 9.

Examples In the following examples, INPUT is a component with the sole output port `in` and the parameter `path`. Instances `y1` to `y4` are all functionally equivalent, as are `z1` and `z2`. Port `in2` of MyComponent is optional and `param2` defaults to `"abc"`.

```
x1 = INPUT(path="x1.csv")
x2 = INPUT(path="x2.pdf")

y1 = MyComponent(in1=x1.in, in2=x2.in, param1=5, param2="abc")
y2 = MyComponent(in1=x1,    in2=x2,    param1=5, param2="abc")
y3 = MyComponent(x1,        in2=x2,    param1=5, param2="abc")
MY_CONST = 5
y4 = MyComponent(array1, x2, param1=MY_CONST)

z1 = MyComponent(in1=array1, param1=5)
z2 = MyComponent(in1=array1.in, in2=null, param1=5)
```

Using type check bypassing and annotations:

```
// Force a PDF (x2) to a CSV port (in1): results in runtime error
err = MyComponent(force in1=x2, param1=MY_CONST)

cond1 = true
cond2 = false
disabled = MyComponent(in1=x1, param1=5, @enabled=cond1&&cond2)

// Not executed: bound to disabled component instance.
```

Name	Type	Values
`@bind`	Component instance name	Let c1 be the component instance name used in the annotation (not a string), and let c2 be the current component instance name where the annotation is declared. Then, the execution of c2 will start only once the execution of c1 has been successfully completed. Note that no data dependencies (output ports of c1 connected to input ports of c2) are needed. In case c1 is not found or is wrapped in double quotas as if a string, a binding error at parse time is triggered. An exception is also triggered if a cycle of dependecies that lead to execution deadlock is declared. This annotation allows for parallel execution between a set of independent components bound for instance to a common component. Null values are silently ignored.
`@enabled`	boolean	`true` (default): the component instance is enabled. `false`: do not execute the component instance. The annotation is propagated to other component instances as follows. First, connections from the disabled instance to optional input ports are removed. If there are connections to mandatory input ports, the target component instance is also disabled.
`@execute`	string	`"changed"` (default): execute the component instance only if its configuration has been changed since the last run or last run was unsuccessful. `"always"`: always execute the component instance, even if configuration has not been changed. `"once"`: execute the component instance only once. On subsequent runs, it is not re-executed even if configuration has been changed.
`@host`	string	Host ID of a remote host on which the component instance is executed, or null/`local` (default) for local host. The value `auto` enabled auto-scheduling. See section 5.12 for details.
`@keep`	boolean	`true` (default): component's output files are stored after usage. `false`: All output files created by the component are removed from the disk after they have been used by downstream components in the script.
`@name`	string	Renames the component instance in the workflow. The instance is available under both the new and original name in the script.
`@par`	record	A mapping from parameter labels onto parameter values; parameters referred in the record instanciate a call to a component. No default.
`@priority`	int	By default every component is set to priority 0. Priorities are compared against each other for those components ready for execution; a higher value means the annotated component will be executed before other components with lower values. Negative values are allowed.
`@cpu`	int	The number of CPUs the component instance gets to allocate. Used with Slurm execution mode.
`@memory`	int	The amount of memory in megabytes the component instance gets to allocate. Used with Slurm execution mode.

Table 9: Component instance annotations that modify the execution logic of the work-

```
always = MyComponent(in1=x1, param1=5,
             @execute="always", @bind=disabled)

// Inserts "name1" into workflow and "renamed" and "name1"
// into AndurilScript name space.
renamed = MyComponent(in1=x1, param1=5, @name="name"+1)
```

Modifying ports attributes Port attributes such as optionality may be modified using port level annotations. Currently we have available the `@require` port annotation which modifies at network construction time, the optional attribute of a port to make it mandatory. This is useful when propagating `@enabled` annotations downstream. Upcoming port annotations include the `@doc` annotation for adding a documentation, and `@type` for specific port type casting.

So far, consider for instance,

```
x1 = INPUT(path="x1.csv")
x2 = INPUT(path="x2.pdf")

isRequired = true

// Assume the second in port in MyComponent is declared as
// optional in component.xml

y1 = MyComponent(x1,                        x2, @enabled = false)
y2 = MyComponent(x1, @require = false       y1)
y3 = MyComponent(x1, @require = isRequired y2)
```

In $y2$ the second in port remains optional and this the disabling from $y1$ takes no effect. However, in $y3$ the second in port is set as mandatory and thus the disabling is propagated from $y1$ to $y3$.

Collections of annotations A set of annotations may be collected under a record, so that component instances with same annotation values may be annotated through the record; this allows for a batch manipulation of annotations. Consider for instance,

```
x1 = INPUT(path="x1.csv")
x2 = INPUT(path="x2.pdf")

ci_annot = record ( execute  = "always",
                    keep     = true,
```

```
                    priority = 10 )

port_annot = record ( require = true )

y1 = MyComponent(x1,               x2, @=ci_annot )
y2 = MyComponent(x1, @=port_annot x2, @=ci_annot, @bind = y1 )
y3 = MyComponent(x1, @=port_annot x2, @=ci_annot, @bind = y1 )
```

Note the `@=` notation for referring to a record with annotation labels and valid values. Also to mention that multiple port annotations may be denoted for each port (despite that we have currently only one port annotation available, in the near future there will be more available).

5.4 Defining composite components: `function`

Composite components, or sub-workflows, enable to break a workflow into smaller and more manageable pieces (see Section 1.2). This makes it possible to create large and maintainable workflows. Composite components are instantiated like regular components. This section describes how composite components are defined. Composite components are defined with the following syntax:

```
function MyFunction(InType1 in1, ..., optional InTypeM inM,
    ParType1 param1, ..., ParTypeP paramP=defaultP)
    -> (OutType1 out1, ..., OutTypeN outN)
{
    ... statements ...
    return record(out1=x1, ..., outN=xN)
}
```

This defines a composite component named `MyFunction` that has input ports named `in1` to `inM`; simple parameters `param1` to `paramP`; and output ports `out1` to `outN`. Types of input ports are given by `InTypeX` and types of output ports by `OutTypeX`. These may be any port types. To define an array type use `Array<T>` in place of `InTypeX` or `OutTypeX` where T is some specific port type. Types of simple parameters `ParTypeX` are listed in Table 1; e.g., `int` is an integer parameter . Input ports may be optional; this is specified using the keyword `optional`. Optional ports must come after mandatory ports. Simple parameters may have default values; these are given as "`ParType param=default`", where `default` is an expression. Also supported are records as parameters, where the parameter type keyword is `record`.

The function must contain a return statement which ends the function call and returns results to the caller. There may be several return statements when the function has if-statements, for example. When the function has multiple output ports, results are returned using a record expression. The expression constructs a record with name-value pairs. Names must correspond to output ports and values must be output ports. A record created using the record expression is similar to the records created by regular component instantiation (see Section 5.3) and it can also be assigned to a variable. If the function has only one output port, the following shorter format may be used:

```
return comp.port
```

Consider a function call such as

```
x = MyFunction(...)
y = MyFunction(...)
```

Here x and y are component instances from which we can refer to the output produced by the execution of each function call.

Example In the following example, the composite component MyFunc computes (m1+m2+m3) · m1 and (m1+m2+m3) · m2 using matrix operations. The input m3 is optional. An numeric bias is added to the matrix sum. After the function definition, the composite component is instantiated two times using different bias arguments. In both instances, m3 is omitted. This can be done in two ways as shown below.

```
function MyFunc(Matrix m1, Matrix m2, optional Matrix m3,
    float bias=0) -> (Matrix prod1, Matrix prod2)
{
    sum = AddMatrix(m1, m2, m3, bias=bias)
    p1 = MatrixProduct(sum, m1)
    p2 = MatrixProduct(sum, m2)
    return record(prod1=p1.product, prod2=p2.product)
}
m1 = INPUT(path="m1.matrix")
m2 = INPUT(path="m2.matrix")
x1 = MyFunc(m1, m2, m3=null)
x2 = MyFunc(m1, m2, bias=1)
```

5.5 Including other files: include

Workflow configuration can be divided into several files to facilitate modularity. Also, reusable items such as function definitions can be placed into their own files and used in

several projects. Including code from another file is done with

```
include "otherfile.and"
```

The argument is a string, so all string operations are available: e.g., `include $ENVVAR+"myfile.and"`. The file name is relative to the main source file, unless an absolute path is given.

5.6 Static conditional processing: if – else

If-statements can be used to conditionally affect workflow construction. The syntax is

```
if expression {
    ... statements ...
}
else {
    ... statements ...
}
```

The expression is evaluated statically and it must use only simple values (Booleans, strings and numbers). It cannot refer to dynamic output results of components. The expression must evaluate to a Boolean. The `else` body may be omitted. The statement blocks may contain any statements.

5.7 Dynamic conditional branches: switch – case

Conditional branches are used to dynamically select alternative routes on the workflow (see Section 1.2). Branches are composed of three elements: a branch component, two or more alternative routes and a join component. Each alternative has a unique name. Branch components are a special type of components: they have a hidden output port that indicates which alternatives are enabled. The hidden port is read by the workflow engine. The manual page of a component indicates whether the component is a branch component.

A conditional branch is placed on the workflow with the following syntax:

```
myBranch = BranchComponent(...)
myJoin = switch myBranch {
    case a1 = MyComponent1(...)
    case a2 = MyComponent2(...)
    ...
    case aN = MyComponentN(...)
    return JoinComponent(a1.port, a2.port, ..., aN.port)
}
```

BranchComponent initiates the branch. It may be any component that can function as a branch component. It defines the set of named alternatives, which are here a1 to aN. There is one case statement for each alternative. Each case statement contains the instantiation of a single component. That component may be a composite component, so the alternatives can contain arbitrary sets of component instances. (Having a single component instantiation simplifies the syntax.)

JoinComponent ends the branch and produces a final result. The outputs of alternatives are available as records named a1 to aN in the return statement. The branch-ending return statement should not be confused with the return statement of a function. The join component is a regular component. However, all input ports of the join component that are connected to outputs of alternatives should be optional, as any of the alternative results may be missing. The join component can assume that at least one alternative is enabled. The results of the join are available to the rest of the workflow as a record named myJoin. Notice that records a1 to aN are not visible outside of the switch statement.

Example In the following example, MatrixCompare is a branch component that takes in a matrix and a numeric threshold and defines three alternatives: equal, less and greater. The alternatives are enabled if the matrix contains one or more elements that are equal to, less than or greater than the threshold, respectively. One, two or three alternatives may be enabled. If the matrix contains elements equal to 5, MyComponent1 is executed and the result is available as the record equal; other alternatives are processed similarily. The join component, Xor, has three optional input ports. The output of Xor is equal to the first input that is present. For example, if equal is not enabled and less and greater are enabled, the less input is returned.

```
matrix = INPUT(path="matrix")
compareResult = MatrixCompare(matrix, threshold=5)
join = switch compareResult {
    case equal   = MyComponent1(...)
    case less    = MyComponent2(...)
    case greater = MyComponent3(...)
    return Xor(equal.port, less.someport, greater.myport)
}
x = OtherComponent(port=join.result, ...) // Use branch results
```

5.8 Native functions

In addition to atomic and composite components, AndurilScript enables to call native functions, which are implemented in Java and are executed during workflow construction. Native functions are executed in the order they are encountered in the AndurilScript program. This is in contrast to workflow execution, in which components may be executed in any order.

Native functions may have any number of inputs but produce only a single output (which may be `null`). Arguments for native functions can be given by position, i.e., without explicitly naming the argument. Some functions, however, have parameters that are given by name.

Standard native functions, accessed through a record named `std`, are listed below. Function signatures are in format `name(arguments)` → `return-type`. The notation `T...` refers to an arbitrary number of unnamed arguments, having type T. The type `any` means any type. In addition to the functions below, there are standard native functions for producing iterators: see Section 5.9.

5.8.1 Generic functions

Signature `std.concat(any..., sep=" ")` → `string`
Description Produces a concatenation of the string representations of the arguments. The given separator is used between the elements.

Signature `std.echo(any..., sep=" ")` → `null`
Description Print each argument to screen. The arguments are separated by the string given by `sep`; the default separator is one space character. The separator argument, if provided, must be named explicitly.
Example

```
x = 5
std.echo("Hello world", x, sep=" ; ") // Prints "Hello world ; 5"
std.echo("Bye") // Printed after the previous message
```

Signature `std.time(value, in="yyyy-MM-dd HH.mm.ss", out="yyyy-MM-dd HH.mm.ss")` → `string`
Description Converts current time (no value) or the given `value` to a string of the `out` format. Values may be given in `in` format, which follows the Java SimpleDateFormat syntax.
Example

```
// Returns 23.10.2013 15:59:25.
time = std.time(value='15.59.25 2013-10-23',
                  in='HH.mm.ss yyyy-MM-dd',
                 out='dd.MM.yyyy HH:mm:ss')
```

Signature `std.convert(boolean/float/int/record, type=string)` → `boolean / float / int / record`
Description Converts the given object to the requested type. Type is the name of the output type: boolean, float, int. For the records, each item is converted independently and a new record is produced for the output.
Example

```
std.convert("100.5", type="int") // =100
// ={'a'=100.5, 'b'=100.0}
std.convert({'a'="100.5",'b'=100}, type="float")
```

Signature `std.recordToString(record, valueSep="=", itemSep=",", keys=true, values=true)` → `string`
Description Converts a record to a string. A key-value pair is separated with `valueSep` if both are included, whereas each two consecutive pairs are separated with `itemSep`. An empty string is produced if keys and values are both false.
Example

```
rec = record(key1="val1", key2="val2")
std.echo("Original record:", rec)
// Prints "Original record: Record: key2 (STRING), key1 (STRING)"

str1 = std.recordToString(rec)
// Returns string "key1=val1,key2=val2"

str2 = std.recordToString(rec, keys=false)
// Returns string "val1,val2"

str3 = std.recordToString(rec, valueSep="->>", itemSep=";")
// Returns string "key1->>val1;key2->>val2"
```

Signature `std.fail(any..., assert=false, sep=" ")` → `null`
Description Produces an error with the message that is constructed as in `std.echo()`. This function can be made conditional if `assert` parameter is used. Statements with `assert=true` are skipped silently and they can be used as invariants to confirm various

conditions.

Example

```
a = 123
std.fail("Invalid A:", a, assert=(a==123)) // OK
std.fail("Halt") // Generates an error that says 'Halt'
```

Signature `std.lookup(string)` → `any`

Description Return the object (variable) whose name is given as a string. If the name is not found, produce an error. This functions makes it possible to find component instances or other variables "dynamically" based on name.

Example

```
myinstance = SomeComponent()
ref = std.lookup("myinstance") // ref is alias for myinstance
```

Signature `std.exists(string, type=string)` → `boolean`

Description Returns a Boolean indicating the existence of an environment variable (`type=env`), a file (`type=file`), an AndurilScript object (`type=object`) of the given name.

Example

```
if (std.exists("HOME_DIR", type="env")) {
   home = $HOME_DIR
} else {
   home = './'
}
```

Signature `std.registerJava(string)` → `function`

Description Register a custom native function that can be used like standard native functions. The sole argument gives the fully qualified Java class name for the class that implements the function. The class must extend

`fi.helsinki.ltdk.csbl.anduril.core.readers.networkParser.functions.NativeFunction`

and must have a public constructor with no arguments. Refer to Java API of `NativeFunction` for details. Implementations of standard native functions are in the aforementioned package and can be used as models.

Example

```
// We register another copy of std.echo
myecho = std.registerJava("fi.helsinki.ltdk.csbl.anduril.core."
  +"readers.networkParser.functions.EchoFunction")
myecho("Hello world", 42)
```

Signature `std.metadata()` → `record`
Description Returns a record with fields

- `instanceName`: component instance for an AndurilScript function call, not null if used within AndurilScript function;
- `Anduril`: engine version;
- `file`: source code AndurilScript pipeline filename;
- `path`: source code AndurilScript pipeline file absolute path;
- `location`: pretty print for location in source code where this std function is called;
- `line`: line location numerical value; and
- `column`: column location numerical value.

This information may prove useful for debugging or reporting.
Example

```
function myFunction() -> (Matrix rnd)
{
    rec = std.metadata()
    std.out("Within body function", rec.instanceName)

    rnd = myComponent()
    return rnd
}

rec = std.metadata()
if (rec.instanceName == null) {
    std.echo(rec.location, "not within body function.")
}

f()
```

5.8.2 File functions

Signature `std.fRead(filename:string | portname:string)` → `string`
Description Returns the contents of a file as a string of up to 1024 Java characters. Input may be a string file name or a reference to an outport from a component instance in a pipeline.

Example

```
// Equivalent calls.
inA = std.fRead("pathToFile/myFilename"))
```

```
inB = std.fRead(INPUT(path="pathToFile/myFilename"))

ci = myComponent()
contentsPortA = std.fRead(ci.portA)
```

Signature `std.nRows(filename:string | portname:string)` → `string`
Description Returns the number of rows of the argument text file. Input may be a string file name or a reference to an outport from a component instance in a pipeline. This function is useful for instance for counting rows in a CSV file.

Example

```
// Equivalent calls.
nRowsA = std.nRows("pathToFile/myCSVFilename"))
nRowsB = std.nRows(INPUT(path="pathToFile/myCSVFilename"))

ci = myComponent()
nRowsPortA = std.nRows(ci.portA)
if (nRowsPortA > nRowsA) {
}
else {
}
```

5.8.3 String functions

Signature `std.substring(str:string, n:integer, m:integer)` → `string`
Description Returns the substring contained in str that is delimited by positions n and m-1. The least value of n is 0 and $n \leq m$.

Example

```
substr = std.substring("012345",1,3)  // delivers "12"
substr = std.substring("012345",0,3)  // delivers "012"
substr = std.substring("012345",1,0)  // error
substr = std.substring("012345",0,0)  // delivers ""
```

Signature `std.length(string/record)` → `integer`
Description Returns the number of characters in a string or the number of elements in a record.

Example

```
strLen = std.length("012345")  // delivers 6
```

Signature `std.strReplace(str:string, [pairs:record|match:string, replace:string], ...)` → `string`
Description Returns input str with *match* occurrences replaced by *replace*. The ellipsis ... may be replaced by as many further match-replace pairs as you wish. Alternatively, the pairs may be encapsulated into records. The extra substitutions will be applied in order from first to last just as if you would call the same function repeatedly. The match may be a Java regular expression.

Example

```
lstr1 = "abcde"
rstr1 = std.strReplace(lstr1,"a","0") // delivers "0bcde"
lstr1 = std.strReplace(rstr1,"0","a") // delivers "abcde"
rstr1 = std.strReplace(lstr1,"z","q") // delivers "abcde"
rstr1 = std.strReplace(lstr1,lstr1,"") // delivers ""
rstr1 = std.strReplace(lstr1,"a","1234") // delivers "1234bcde"
rstr1 = std.strReplace(lstr1,"a","1234","d","4321") // delivers
                                                    "1234bc4321e"
```

Signature `std.quote(string, type=string)` → `string`
Description Adds language specific escape sequences to the given string so that it can be embedded to the source codes based on the language. Possible types are: 'anduril' for legal AndurilScript instance names, 'html' for HTML escapes, 'latex' for LaTeX escapes, and 'url' for `application/x-www-form-urlencoded` values.
Example

```
sampleId     = "ID_123"
sectionTitle = std.quote(sampleId, type="latex")+ // "ID\_123"
               " sample"
section      = LatexCombiner(sectionTitle = sectionTitle)
```

5.8.4 Numeric functions

Signature `std.mod(n:integer, m:integer)` → `integer`
Description Returns the reminder of n divided by m.

Example

```
val = std.mod(0, 2)  // delivers 0
val = std.mod(1, 2)  // delivers 1
val = std.mod(2, 2)  // delivers 0
val = std.mod(3, 2)  // delivers 1
```

Signature `std.pow(n:float, m:float)` → `float`
Description Returns the value of n raised to the power of m.

Example

```
val = std.pow(1, 2)  // delivers 1
val = std.pow(2, 4)  // delivers 16
val = std.pow(2.5, 2)  // delivers 6.25
```

5.9 Looping over iterators: `for`

Sometimes, it is necessary to produce iterative structures in AndurilScript such as the following:

```
x1 = SomeComponent(k=1)
x2 = SomeComponent(k=2)
...
xn = SomeComponent(k=n)
```

For large n, this becomes tedious and error-prone. This motivates the introduction of static looping. The previous example is written more compactly as:

```
array = record()
for i: std.range(1, n) {
  array[i] = SomeComponent(k=i)
}
```

Here, `i` is an index variable that gets assigned a new value for each iteration. `std.range` is a standard native function that produces an iterator for the integer range n to m. For $n < m$ it will produce a monotonically increasing sequence, whereas for $n > m$ it will procude a monotonically decreasing sequence. A third, optional integer argument d defines the distance between two consecutive values in a sequence. If not declared, the default distance is 1. When $d < 0$ and $n > m$, the sequence will decrease by d at each

consecutive value and for $d > 0$ and $n < m$, the sequence will increase by d at each consecutive value. For $n = m$, $d = 0$ or $abs(d) > abs(m - n)$ (absolute values), only the value of n is provided.

For $n = 1$ and $n < m$, component instances are placed into a record data structure (essentially, a hash table) and can be accessed with `array[1]` to `array[n]`. Component instances names in the workflow are `array_1` to `array_n`, but these names are not visible at AndurilScript level.

An alternative to the use of records is to use `@name` annotations (see Section 5.3) and the std.lookup (see Section 5.8) standard function. In the following example, the results of the previous iteration are used for the next iteration, unless $i = 1$.

```
for i: std.range(1, n) {
  if (i == 1) x = SomeComponent(@name="x"+i)
  else {
    prev = std.lookup("x"+(i-1))
    x = SomeComponent(inPort=prev.outPort, @name="x"+i)
  }
}
```

The `@name` annotation is used to assign a unique name for each component instance, as required by the workflow structure. When records are used, names are generated automatically. The names `x1` to `xn` refer to component instances corresponding to "absolute" indices, while `x` refers to the newest instance that is replaced on every iteration. Thus, after the loop is finished, the last component instance is available as both names `xn` and `x`. Results of the *k*'th iteration are accessed using std.lookup, which allows to map a string to an object. The string can, of course, be constructed by concatenating an index to a base name.

The general syntax of the for-loop is as follows:

```
for Name1, ..., NameK: iterator expression {
  ... statements ...
}
```

An iterator produces a vector of K values on each iteration that are assigned to Name1 to NameK. The iterator is produced by a call to a native function (either standard or customly registered); also, records can be iterated directly. In the example above, `std.range` produces vectors of length one.

Using `std.registerJava` (see Section 5.8), it is possible to introduce custom native functions that produce iterators. These functions must return a value of type ITERATOR

that extends the class

`fi.helsinki.ltdk.csbl.anduril.core.readers.networkParser.value.IteratorValue`.

Below, standard native functions producing iterators, as well as the record iterator, are described. The lengths of vectors produced by iterators are shown in parentheses.

Signature `record` → `iterator(2)`
Description A record can be iterated over as (key, value) pairs. No native function call is needed: a record is a valid iterator expression by itself.
Example

```
rec = record(x1=1, x2=2, x3=3)
for key, value: rec {
  // Produces ("x1", 1), ("x2", 2), ("x3", 3)
}
inst = SomeComponent()
for outPortName, portValue: inst {
  // Iterate over out-ports of an instance
  x = OtherComponent(in=portValue, @name="x"+outPortName)
}
```

Signature `std.range(low: int, high: int, step: int)` → `iterator(1)`
Description A basic iterator that produces numeric values from the low bound to the high bound (inclusive) where two consecutive values are separated by a number of units specified by step.
Example

```
// Iterates over 1, 2, ..., 10
array = record()
for i: std.range(1, 10) {
  array[i] = SomeComponent(k=i)
}
```

Signature `std.split(string, string)` → `iterator(1)`
Description Provides a string tokenizer for the first argument. Tokens are separated with delimiters of the second argument. The default delimiter is a white space sequence.
Example

```
// Iterates over a, b, and c.
for l: std.split("a b\t  c") {
  std.echo(l)
}
for l: std.split("a,b,c", ',') {
```

```
  std.echo(l)
}
```

Signature `std.itercsv(string)` → `iterator(1)`
Description Iterate over row of a CSV (tab-delimited) file that has column headers. The name of the CSV file is given as the sole argument. Each iteration progresses over one row and produces a record that binds column names to the values on the current row. The CSV is a static file: it must be available at parse time and is not dynamically generated by Anduril components. Data types of each column are inferred from the CSV file, so numeric columns are produced as integers or floats. The data types produced for each column are consistent: a string column does not yield numeric values even if some individual values could be interpreted as numeric. However, NA values in the CSV file produce null values into the records.
Example

```
/*
table.csv contains:
Col1   Col2   Col3
x1     5      -9.2
x2     NA     7
3      2      NA
*/
array = record()
for row: std.itercsv("table.csv") {
  // Produces record(Col1="x1", Col2=5, Col3=-9.2),
  //          record(Col1="x2", Col2=null, Col3=7.0),
  //          record(Col1="3", Col2=2, Col3=null)
  if (row.Col2 != null && row.Col3 != null) {
    array[row.Col1] = SomeComponent(p1=row.Col1,
      p2=row.Col2, p3=row.Col3)
  }
}
```

In this example we test for null values in Col2 and Col3. The value of Col1 on the third row is interpreted as a string instead of an integer since the type of Col1 is inferred to be string.

Signature `std.iterdir(string)` → `iterator(1)`
Description Iterate over the files in the directory coded in the argument. If the argument is a file, then that only file is provided to the body of a for loop that embodies the iterator. Each element provided by the iterator includes a reference to the file name and

a reference to the full path (see the following example for Linux-like file systems).
Example Assume

```
> ls ./myDataDirectory
data1.csv data2.csv data3.csv
```

Then

```
for file: std.enumerate(std.iterdir("./myDataDirectory")) {
  std.echo(file.name,file.path)
}
```

echoes a text like

```
data1.txt  /home/myAccount/myDataDirectory/data1.csv
data2.txt  /home/myAccount/myDataDirectory/data2.csv
data3.txt  /home/myAccount/myDataDirectory/data3.csv
```

Signature `std.enumerate(iterator...)` → `iterator(1+K1+...+Kn)`
Description Return a higher order iterator, taking another iterators(s) as argument, that attaches numeric indices to the values produced by child iterator(s). This is useful when the argument iterator does not have a natural unique key, as is common with CSV iterators. Indices start from 1. When multiple iterators are given, iteration continues until one of the child iterators signals a stop condition. The length of the vector is the sum of child vector lengths (Ki) plus one, for the index. The first element of the vector is the index, and the following elements are the concatenated results of child iterators.
Example

```
for idx, i: std.enumerate(std.range(20, 29)) {
  // Produces (1, 20), (2, 21), ..., (10, 29)
}
for idx, i1, i2: std.enumerate(std.range(20, 29),
                               std.range(30, 35)) {
  // Produces (1, 20, 30), (2, 21, 31), ..., (6, 25, 35)
}
```

5.10 Dynamic `for`-loop and `include` statement

It is possible to access contents of component output ports in an AndurilScript program during the evaluation of the program. This dynamic facility is supported by the iterators

`std.itercsv` and `std.iterdir` as well as the `include` statement. This enables dynamic for-loops, in contrast to static ones introduced in Section 5.9, and dynamic generation of AndurilScript source. Custom iterators and native functions also have access to the facility.

The syntax is a straightforward extension of the static counterparts. The string arguments for `std.itercsv`, `std.iterdir` and `include` are replaced with a reference to a component instance output port. The following demonstrates the use of these forms.

```
// Produces CSV output, directory output and AndurilScript output
inst = SomeComponent()
for row: std.itercsv(inst.outputCSV) {
    // Loop over contents of CSV output
}
for rec: std.iterdir(inst.outputDirectory) {
    // Loop over contents of directory output
}
// Include generated AndurilScript source code
include inst.outputAndurilScript
```

There are some restrictions and considerations when using the dynamic facility.

- The component instance whose output is used dynamically must not have `@enabled=false` or `@keep=false` annotations.
- Component instances introduced before a dynamic for-loop or `include` statement may be executed several times during the same run if they have `@keep=false` or `@execute="always"` annotations.

5.11 Array data types

Commonly, workflow data input consists of a list of similar file items, such as a list of measurement files for independent samples. To this end, Anduril component model supports array data types, which are homogeneous ordered collections of elementary files (or directories). In component HTML manual pages this is indicated with `Array<T>`, where T is an elementary file data type. Anduril arrays are associative: they are indexed using string keys. Integer keys are automatically converted to strings. Formally, an array of type T is an ordered collections of (key, value) pairs where keys are unique strings and values are files of type T. Arrays are implemented using index files and array operations generally do not make extra copies of element files for efficiency. Arrays are passed as input to components and functions like regular types. AndurilScript records and array data types are similar data structures – ordered associative arrays – but the difference is

that records are in memory and arrays are on disk. Array-specific operations include the following:

- Constructing arrays is done implicitly using AndurilScript records or by using components that produce arrays. Alternatively, it is possible to explicitly call the standard native function `std.makeArray`. It is also possible to manually write array index files (see Section 6.3.2 for format) and import them using the INPUT component.
- Constructing an array from folder contents is done using the builtin component Folder2Array.
- Arrays can be combined and filtered using the builtin component ArrayCombiner. It enables taking unions and intersections of keys and filtering keys using regular expressions.
- Extracting a known key is done using record access forms `inst.arrayPort.key` or `inst.arrayPort["key"]`, where `inst` is a component instance producing an array as output.
- Array entries can be dynamically iterated over using `std.iterArray`.

In the following example, an example component ArrayConsumer takes an array as input and produces an array as output. In `inst1`, an implicit array is constructed from a record. The produced array is passed to `inst2`. Assuming ArrayConsumer always produces an array with key `key1`, the file corresponding to this key is extracted and passed to `inst3`. The array produced by `inst2` is filtered by selecting only keys that start with `key`. The second half demonstrates dynamic iteration of an array. Each element of `inst2` is passed to ComponentX and its output is gathered into the `values` record. Finally, an explicit array is constructed from this record. This corresponds to the higher order function `map(f, A)` that applies function `f` (ComponentX) to each element of array `A` (`inst2.arrayOutput`).

```
// x1, x2 and x3 are atomic input files
inst1 = ArrayConsumer(arrayInput={x1, x2, x3})
inst2 = ArrayConsumer(inst1.arrayOutput)
inst3 = NonArrayComponent(inst2.arrayOutput["key1"])
filtered = ArrayCombiner(inst2, subset="key.*")

values = {}
for rec: std.iterArray(inst2.arrayOutput) {
    values[rec.key] = ComponentX(inst2.arrayOutput[rec.key])
}
valueArray = std.makeArray(values)
```

Native functions related to arrays are as follows:

Signature `std.iterArray(string or output port)` → `iterator(1)`
Description Iterate over key-value pairs of an array. The input can be a string path to an array directory (static iteration) or the name of an array output port (dynamic iteration). The iterator produces records with the fields `key` (string key) and `file` (element file path).

Signature `std.makeArray(...)` → `output port`
Description Produce an array from any number of records, existing arrays, or atomic files. The returned value is an output port for a component instance that produced the array. The mechanism for producing the array is implementation specific, but the output port always captures the final result. If existing arrays are given, their contents are appended to the result array using union key semantics; this enables to use this function like ArrayCombiner supporting arbitrary number of input arrays. If there are multiple key-value pairs with the same key, the first is selected. Calling this function explicitly is often not necessary as arrays are constructed implicitly from records in component calls.
Example

```
// x1, ..., xN are atomic input files
a1 = std.makeArray({x1, x2, x3}, key1=x4, record(key2=x5))
// a1 = 1->x1, 2->x2, 3->x3, "key1"->x4, "key2"->x5
a2 = std.makeArray(a1, a1, a1, {"key3"=x6})
// a2 = 1->x1, 2->x2, 3->x3, "key1"->x4, "key2"->x5, "key3"->x6
```

5.12 Executing components on remote hosts

Anduril supports executing component instances on remote hosts to improve parallelization and take advantage of cluster or distributed computing facilities. The host can be selected individually for each component instance. There are two modes of remote execution: (1) with and (2) without a shared file system. A non-shared system may initially be faster to set up, but shared mode reduces file transfers between hosts and is therefore recommended. Authentication must be done non-interactively, such as using SSH public keys (see Section 3.1.5 for general tips on setting up SSH). Currently, only the Anduril command line interface supports this mechanism. The remote system is currently assumed to be a Unix system.

Remote execution is enabled using these steps:

1. Copy `bin/anduril-remote` to remote host(s) along their path. Also ensure that relevant components and their external resources are installed. Anduril engine

itself does not need to be installed on remote hosts.

2. Define a `hosts.conf` file that describes remote hosts. Invoke `anduril run` with the argument `--hosts hosts.conf`.
3. Define `@host="hostID"` annotations (see Section 5.3) for component instances to be executed on remote hosts. The special value `@host="auto"` allows an auto-scheduler to select the host.

All file system resources needed by components, such as their resource bundles, must be present on both file systems (which may be a shared one). The hosts configuration file defines a number of path mappings from the local to remote file system. In shared mode, also input files imported into the workflow using INPUT components may need to be mapped, whereas in non-shared mode they are copied by the engine. Anduril engine will report missing mappings so it may be easiest to start with a minimal mapping and add entries as needed. All paths in the configuration file must be absolute.

Concurrency is controlled with two parameters. Each host (including local host) may have a maximum number of concurrent component executions, called slots. Total concurrency (processes in all hosts) is controlled with the thread limit of the workflow engine (assigned with `--threads` command line argument). When slot limits are defined, `--threads` should be set to the sum of all slots for maximum concurrency.

In non-shared mode, there is one master execution directory on the local host, and mirrored execution directories on remote hosts that contain a subset of component instance outputs. All output appears eventually in the master directory. Remote mirror directories can be cleared manually by user if necessary in non-shared mode; Anduril considers them temporary with a life time of one workflow execution.

5.12.1 Remote host configuration file

The hosts configuration file may have one or more remote host definitions. It has a `key=value` format with keys described in table 10. Hosts are separated by a blank line. Long lines can be split by having a backslash (\) at the end of line. Lines starting with hash (`#`) are ignored.

Configuration entries `RemoteExecute`, `CopyLocalRemote` and `CopyRemoteLocal` define patterns for executing commands on the local host that interact with the remote host. In these patterns, the token `${REMOTE_HOSTNAME}` is replaced with the value of `HostName`, `${REMOTE_PORT}` with `RemotePort` and `${REMOTE_USER}` with `RemoteUser`. In `RemoteExecute`, `${COMMAND}` is replaced with current arguments, and in the copy commands, `${LOCAL_PATH}` and `${REMOTE_PATH}` are replaced with

Key	Description
HostID	Host ID that is used in @host annotations. The local host has the special value local.
HostName	IP or DNS address or SSH alias of the host.
RemoteExecutionDirectory	Location of execution directory on remote host.
IsSharedFileSystem	true if file systems are shared between local host and the current remote host, false otherwise (default: false).
PathMapping	List of LOCAL=REMOTE pairs that define how to map local file paths to remote file system, delimited by whitespace.
RemotePort	TCP/UDP port on remote host (default: 22).
RemoteUser	User name on remote host (default: not defined).
Slots	Maximum number of concurrent component executions on the host (default: unlimited). Can also be specified for local host.
Wrapper	If defined, the name of a wrapper script on remote host that is used for component launching (default: not defined). The wrapper is given the launch command as arguments. This enables, e.g., interacting with a job queue.
RemoteExecute	Pattern that defines the command for executing a remote command (default uses ssh). See below.
CopyLocalRemote	Pattern that defines the command for copying a local file to remote host (default uses rsync over SSH). Only needed in non-shared mode.
CopyRemoteLocal	Pattern that defines the command for copying a remote file to local host (default uses rsync over SSH). Only needed in non-shared mode.

Table 10: Key descriptions for host definition file. Only entries with no default are mandatory.

with local/remote paths. In addition, ${LOCAL_DIR} and ${REMOTE_DIR} are replaced with local/remote directory names, i.e., ${LOCAL_DIR} is the parent directory of ${LOCAL_PATH}. The defaults are (omitting newlines):

```
RemoteExecute = ssh -p ${REMOTE_PORT} -o BatchMode=yes
    ${REMOTE_HOSTNAME} ${COMMAND}
CopyLocalRemote = rsync -a --exclude='**/.*' --port=${REMOTE_PORT}
    ${LOCAL_PATH} ${REMOTE_HOSTNAME}:${REMOTE_DIR}/
CopyRemoteLocal = rsync -a --port=${REMOTE_PORT}
    ${REMOTE_HOSTNAME}:${REMOTE_PATH} ${LOCAL_DIR}/
```

5.12.2 Example

Consider an example with two remote hosts in addition to local host. Host remote1 shares the execution directory with the local host, but has local copies of resource bundles and workflow data files. Host remote2 has no shared file system. It does

not have pre-installed copies of workflow data files because they are copied explicitly when needed; instead, it has a writable local directory (`/home/user/local/data`) into which the data files are copied. Directory structures are as follows:

```
local:
/usr/share/bundle      (resource bundle containing components)
/home/user/data        (data files for workflow)
/home/user/execute     (local execution directory)

remote1:
/usr/local/bundle         (local bundle installation)
/home/user/localdata      (local copy of data files)
/mnt/fs/home/user/execute (shared execution directory)

remote2:                  (no shared file system)
/home/user/local/bundle
/home/user/local/data
/home/user/local/execute
```

Hosts configuration file is as follows. Notice that it is not necessary to define an explicit path mapping between execution directories, only for other resources. In this example, we limit the concurrency of the local host to one component execution, while remote hosts can have up to four concurrent executions.

```
HostID = local
Slots = 1

HostID = remote1
HostName = 192.168.1.1
RemoteExecutionDirectory = /mnt/fs/home/user/execute
IsSharedFileSystem = true
Slots = 4
Wrapper = trivial-wrapper
PathMapping = /usr/share/bundle=/usr/local/bundle\
              /home/user/data=/home/user/localdata

HostID = remote2
HostName = host2.example.com
RemoteExecutionDirectory = /home/user/local/execute
IsSharedFileSystem = false
Slots = 4
PathMapping = /usr/share/bundle=/home/user/local/bundle\
              /home/user/data=/home/user/local/data
```

For illustration, `remote1` uses a trivial wrapper that simply launches the command given:

```
#!/bin/sh
"$@"
```

An AndurilScript workflow using these three hosts would look like follows:

```
x1 = Component1() // local host
x2 = Component2(x1.out,          @host="remote1")
x3 = Component3(x1.out, x2.out, @host="remote2")
x4 = Component4(x3.out,          @host="auto")
```

This is executed as follows. First `x1` is executed locally. Then `x2` is invoked on `remote1`; no file copy operations are needed because `remote1` shares the execution directory with the local host. Before `x3` can be executed, the outputs `x1.out` and `x2.out` must be copied to the mirror execution directory on `remote2`. After `x3` finishes, `x3.out` is copied to the master execution directory on the local host. Finally, `x4` is executed on a host selected by auto-scheduler.

Part II
Anduril for Developers

The following sections are targeted at developers who are interested in implementing new components, data types or bundles for Anduril, or extending existing ones.

6 Implementing components

A component is an executable that reads input files, processes data and writes output files. Because the only requirement is the ability to read and write files, components can be written in any language. To make implementing components more convenient, support packages that handle common component tasks are available for selected languages.

The external interface of a component is specified using an XML file called *descriptor* file. The interface of a component consists of input and output ports and simple parameters. Special components such as branches have additional elements. The descriptor also specifies how the component is to be executed; for example, Java and R components are executed in different ways.

Documentation is an integral part of the descriptor files, as the component manual pages are generated based on descriptor files. All parts of the component interface can and should be documented.

6.1 Directory structure

Each component is stored in a directory with a well-defined structure. See Figure 14 for an overview on directory structure. The descriptor file is located in `component.xml` (see Section 6.2 for the file format). The `testcases` directory contains component black box test cases (see Section 6.5 for details).

In addition, the component directory may contain any other file needed for component execution or documentation. These may include: executable files such as R sources; additional data files needed for execution; and example output files (e.g., PDFs) for documentation.

6.2 Descriptor XML files

The descriptor must be found in the file `component.xml` under the directory of the component. Figure 15 specifies the syntax of descriptor files. Note that the order of

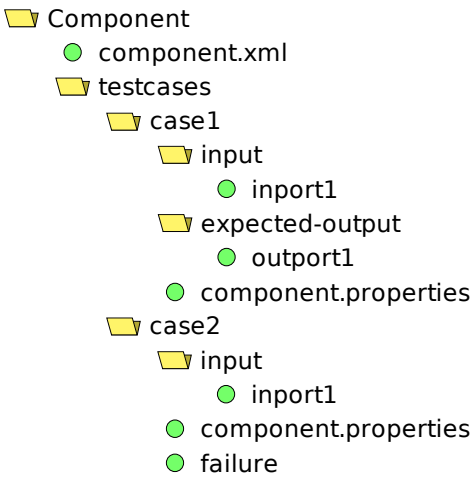


Figure 14: File organization of components.

elements is significant for the root element `component`. Table 11 gives a description of XML elements. Figure 16 contains a template file, while Figure 18 contains a sample descriptor file.

Human-readable documentation is written in `doc` elements. Most elements can hold `doc` elements. The `doc` elements may contain XHTML code, but this should be used only when necessary (*e.g.*, for images). Otherwise, plain text should be used. To ease formatting, two consecutive newlines in a `doc` element means a paragraph break.

Input ports, output ports and parameters are defined using the elements `inputs`, `outputs` and `parameters`, respectively. All these are optional. Individual ports and parameters are child elements of these aggregate elements.

6.2.1 Launchers

Launchers are responsible for executing components. As components can be written in a variety of languages, the Anduril core provides several launchers. For example, R components are executed with the R launcher. There is also a generic Bash launcher.

In the descriptor file, the launcher is identified by a name (e.g. "`R`" for the R launcher). Launchers are also given arguments that tell how the launcher should execute the component. For example, the R launcher needs to know the location of the R source file. Arguments are key-value pairs. There may be several launchers for a component, although usually there is one. All launchers and their arguments are listed in Table 12.

```
component := name version doc instance-class? author* category*
             launcher+ requires* type-parameters inputs? outputs?
             parameters? choices? delegates?

name := STRING
version := STRING
doc := STRING
instance-class? := STRING
author <email?> := STRING
credit <email?> := STRING
category := STRING

launcher <type> := argument*
argument <name value> := EMPTY

requires <name? URL? optional? version? type?>
    := STRING | resource+
resource <type> := STRING

type-parameters := type-parameter*
type-parameter <name extends?> := doc?

inputs := input*
input <name type optional? array?> := doc?

outputs := output*
output <name type array?> := doc?

parameters := parameter*
parameter <name type default?> := doc?

choices := choice*
choice <name> := doc?
```

Figure 15: Syntax of descriptor XML files. The expression `name <attr> := child` specifies an XML element called `name` that has the attribute `attr` and a child element called `child`. If `<...>` is omitted, the element does not have attributes. For attributes and child elements, `x?` means `x` is optional and may occur zero or one times; `x*` means `x` is optional and may occur any number of times; `x+` means `x` is mandatory and may occur one or more times; `x y` means the element `x` must occur before `y`; and `x | y` denotes mutually exclusive alternatives. For elements, `STRING` means the element has character content and `EMPTY` means the element is empty.

Element	Description
`name`	Name of the component.
`version`	Version number, with format `1.0` or `1.0.0` or `1.0.0.0`.
`doc`	The main documentation that describes the purpose and functionality of the component.
`instance-class`	Only used for special internal components that are run inside the same Java Virtual Machine as the workflow engine. This gives the class name of such components.
`author`	Name and optionally the email address (as an attribute) of an author. There may be several `author` elements.
`credit`	Name and optionally the email address (as an attribute) of an acknowledged contributor. There may be several `credit` elements.
`category`	A category that the component belongs to. There may be several `category` elements.
`requires`	Describes an external dependency of the component. This may be a free-format string or, if the `type` attribute is given, a machine-readable dependency. Dependency on an R package is given with type `R-package`. Dependency on a Java JAR file located in the bundle `lib/java` directory is given with type `jar`.
`launcher`	Specifies the launcher. The `type` attribute identifies the launcher type.
`launcher` → `argument`	Passes a key-value pair to the launcher that gives details on how the component should be executed. Argument name and value are given with the attributes `name` and `value`, respectively.
`input`	Input port. Name and type of the port are given using the attributes `name` and `type`. Type must be a port type. If the `optional` attribute is given, it indicates whether the port is optional. Legal values are `true` and `false`. The attribute `array` specifies whether the port is an array port (see 6.2.4) (default: `false`).
`output`	Output port. The format is similar to input ports except the `optional` attribute is not present.
`parameter`	Simple parameter. Type must be a simple type (see Table 1). If the `default` attribute is given, it is the default value for the parameter. The value must match the type of the parameter.
`choice`	For branch components, each `choice` element names an alternative execution route.

Table 11: Description of selected XML elements. The elements are in the order they appear in the XML file. The notation `x` → `y` means that `x` is a parent element of `y`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<component>
    <name></name>
    <version>0.1</version>
    <doc>
    </doc>
    <author email="">Your Name</author>
    <category>XXX</category>
    <launcher type="">
        <argument name="" value="" />
    </launcher>
    <requires></requires>
    <inputs>
        <input name="" type="">
            <doc></doc>
        </input>
    </inputs>
    <outputs>
        <output name="" type="">
            <doc></doc>
        </output>
    </outputs>
    <parameters>
        <parameter name="" type="" default="">
            <doc></doc>
        </parameter>
    </parameters>
</component>
```

Figure 16: Template `component.xml` file.

Launcher	Arguments
`bash`	`file`: location of Bash script. `source`: additional source file names (optional; for manual page).
`java`	`class`: fully qualified name of the Java class that implements the component. `source`: source file names (optional; for manual page).
`lua`	`file`: location of Lua source file. `source`: additional source file names (optional; for manual page).
`matlab`	`file`: location of Matlab source file. `source`: additional source file names (optional; for manual page).
`octave`	`file`: location of Octave source file. `source`: additional source file names (optional; for manual page).
`perl`	`file`: location of Perl source file. `source`: additional source file names (optional; for manual page).
`python`	`file`: location of Python source file. `source`: additional source file names (optional; for manual page).
`R`	`file`: location of R source file. `source`: additional source file names (optional; for manual page).

Table 12: Launchers provided by Anduril core. For each launcher, the name and arguments are given. For launchers that take a source file argument, the source file is automatically added as a source file for the manual page; `source` elements are needed only for additional sources.

6.2.2 Branch components

Branch components define two or more alternative execution routes using `choice` elements. The name of the alternative is given using the `name` attribute. The component has a special output port, `_choices`, that is used to signal which alternatives are enabled. The `_choices` file contains one enabled alternative per line. There must be at least one line as at least one alternative must be enabled. For example, a branch component that enables alternatives `alt1` and `alt2` would write the following write to `_choices`:

```
alt1
alt2
```

6.2.3 Type parameters

Type parameters are given with `type-parameters` elements. The `name` attribute gives the name of the parameter; this can be used for input and output port types in place of a concrete type. If the `extends` attribute is present, it refers to a concrete type that the type parameter must extend. Figure 17 shows an example on defining type parameters.

```
<component>
  ...
  <type-parameters>
    <type-parameter name="T1" extends="CSV" />
    <type-parameter name="T2" />
  </type-parameters>
  ...
  <input name="in1" type="T1" />
  ...
  <output name="out1" type="T1" />
  <output name="out2" type="T2" />
  ...
</component>
```

Figure 17: Example of defining type parameters in a descriptor file. The generic component has two type parameters, `T1` and `T2`. The concrete type assigned to `T1` must be a subtype of `CSV`, while any concrete type can be assigned to `T2`.

6.2.4 Declaring array ports

The `array` attribute of `input` and `output` elements defines whether the port is an array. The attribute has three legal values: `false` (is not array; default), `true` (is array) and `generic`. The `generic` value is only legal for ports that use a generic type (type parameter). It indicates that the port may be an array or non-array; the ”array-ness” is a parameter. The engine infers whether the port is an array depending on context in the workflow.

6.2.5 Multifiles

In addition to plain files and directories as inputs and outputs, Anduril supports *multifiles* that are conceptually a mixture between files and directories. A multifile consists of a primary file that must always be present, and any number of optional auxiliary files that are located in the same directory and have the same basename but a different file extension. In the file system, a multifile is referred to by the path of the primary file and auxiliary files can be located by iterating over files with the same basename. Example: `sample.bam` is a primary file and `sample.bam.bai` and `sample.md5` in the same directory are auxiliary files. Multifiles are used when the presentation type of a data type is declared to be `multifile` in `datatypes.xml` (see Section 7). A common use case for multifiles is file formats that consist of a main file and a number of index or annotation files.

Note Currently, multifiles are not supported in remote execution with non-shared execution directories.

6.2.6 Defining component requirements

External requirements of components are defined with `requires` elements in `component.xml`. This enables automatic installation of (some) software resources using the built-in InstallRequirements component. Each requirement has a type, such as R package or JAR file. If the type is omitted, the element documents the requirement but installation must be done manually. Some requirements may have alternative types, such as both an Ant build file and a make file: if one method fails or is not available, the others can be used.

For simple requirements with no alternative types, an inline format can be used. To denote alternative resource types, child `resource` elements must be used. Syntax for inline (first) and extended (second) formats:

```
<requires type="TYPE" name="NAME" optional="true/false"
   URL="http://...">CONTENT</requires>

<requires name="NAME" optional="true/false" URL="...">
  <resource type="TYPE1">CONTENT1</resource>
  <resource type="TYPE2">CONTENT2</resource>
  ...
</requires>
```

Pairs of TYPE, CONTENT define the actual requirements. Interpretation of CONTENT depends on TYPE. Supported types are listed in Table 13.

The following attributes are all optional. `name` gives a human-readable name for the requirements; it can often be inferred from resource codes. If `optional="true"`, the resource is only needed for some execution paths in the component. `URL` gives a human-browsable WWW address for the resource; it is not used for downloading.

Example The following elements define dependencies for a bundled JAR file, R package Hmisc and optionally for Bioconductor package affy. There are custom installation scripts for Ant and make; only one of them is used, depending on user preferences.

```
<requires type="jar">jarfile.jar</requires>
<requires type="R-package">Hmisc</requires>
<requires type="R-bioconductor" optional="true">affy</requires>
```

Type	Description
`ant`	Ant build file (`build.xml`) located in the component directory. Content gives the build target (default target for empty content).
`DEB`	Name of a Debian installation package.
`jar`	Bundled Java JAR package, located in the `lib/java` directory of resource bundles. Components must declare their JAR requirements. If the resource is `*`, all JARs are included.
`make`	Makefile (`Makefile`) located in the component directory. The description gives the make target (default target for empty content).
`manual`	No automatic installation. The type can also be omitted.
`R-package`	R package name. Installed from CRAN, or if not found there, Bioconductor.
`R-bioconductor`	R Bioconductor package. This must be used instead of `R-package` if there is a CRAN package with the same name. This is also more efficient because CRAN is not queried during installation.

Table 13: Requirement types and their interpretations in `requires` elements.

```
<requires name="Custom script">
  <resource type="ant">my-ant-target</resource>
  <resource type="make">my-make-target</resource>
</requires>
```

6.3 Component execution

A component is executed by the following steps:

1. Workflow engine prepares all necessary information for component execution in a *command file*. This is a simple properties file that contains file names associated to input and output ports and values of parameters.
2. A launcher is used to invoke the component (see Section 6.2.1). The path of the command file is passed as an argument.
3. The component parses the command file, processes input files and writes output files. The component returns an exit status that indicates whether execution was successful.

In practice, components are not written from scratch but rather using language-specific component frameworks that handle common tasks such as parsing command files and handling common file formats. See Section 6.4 for details on component frameworks.

When a component is executed, the present working directory (PWD) is set to the component directory that contains `component.xml` and other component files. This

provides easy access to auxiliary data files that may be distributed with the component.

There is a file (or a directory) associated to each input and output port. These are allocated by the workflow engine. The component may assume that input files are present and readable if their file name is given in the command file. For unconnected input ports, the file name is empty. The component may assume that output files can be written to and do not exist before execution.

All components have special output ports for error and log messages. These enable components to pass messages to the workflow engine, which displays them to the user. Components may also write to the standard output and standard error streams, which are also captured by the workflow engine.

6.3.1 Command files

Normally, components do not directly interact with the physical structure of a command file. Rather, the language-specific framework provides convenient access to the command file.

The command file contains file names associated to input and output ports as well as values for simple parameters. Also, there is metadata such as the name of the component instance in the workflow and the name of the workflow engine that invoked the component.

An example command file is as follows. The order of entries is arbitrary. Input files are given with `input.PORTNAME=FILENAME` and output files with `output.PORTNAME=FILENAME`. Parameter values are given with `parameter.NAME=VALUE`.

```
metadata.instanceName=myInstance
metadata.engine=Anduril 1.0
input.m1=/home/user/workflow/m1.csv
input.m2=/home/user/workflow/m2.csv
input.m3=
output.sum=/home/user/execute/myInstance/sum.csv
output._errors=/home/user/execute/myInstance/_errors
output._log=/home/user/execute/myInstance/_log
parameter.bias=0
```

6.3.2 Array data type implementation

Array data types (see Section 5.11 and 6.2.4) are implemented using a tab-delimited index file that contains string keys and file paths in an ordered collection. The command file contains two entries for each array port. The regular entries `input.PORTNAME` and

`output.PORTNAME` specify a directory that contains the array contents. In addition, special pseudo-port entries `input._index_PORTNAME` and `output._index_PORTNAME` specify paths to the index file. The pseudo-port entries must always be used to access the index file, as its path may depend on context.

Components that write arrays must create the output array directory before use. This directory can be used to write array element files. File names starting with underscore (_) are reserved.

The index file is a CSV file that follows the general form of Section 4.1. The index always contains two columns: Key and File (in capital). Key gives the element string key and File gives the path to the file itself. The path may be relative, in which case it is relative to the directory containing the index file. In future Anduril versions, new columns may be added to the index file.

Array contents may span multiple directories. A component that takes an array as input and writes one as output may replace some elements of the input array by writing replaced items to its output directory. The output index is then a mixture of files located in several directories.

When an input array is produced manually and imported into the workflow using the INPUT component, the array should consist of a directory that contains an index file under the name `_index`. The directory may also contain files whose paths are relative in the index. The name of the directory, not the name of the index file, is used to import the array using INPUT.

An example index file is below. The `relative_file*` files must be located in the same directory as the index file.

```
Key     File
key1    /home/user/data/file1
key2    /home/user/data/file2
42      /usr/share/file3
rel1    relative_file1
rel2    relative_file2
```

6.4 Component frameworks

6.4.1 Framework for R

There is an R package `componentSkeleton` that provides command file parsing, error and log message writing and other functionality. The details are documented in the

package. The functions of the package have R documentation. For example, to see help for the function `main`, load the package in R using `library(componentSkeleton)` and then type `?main`.

6.4.2 Framework for Java

For Java there is a similar helper package in `fi.helsinki.ltdk.csbl.anduril.component`. Components generally extend the class `fi.helsinki.ltdk.csbl.anduril.component.SkeletonComponent`.

6.4.3 Framework for Matlab

Matlab core functions for command file parsing, error and log message writing are in the `matlab` directory. Change your working directory there and find the documentation for each function by providing the function name to the command `help`. The directory is automatically added to Matlab path when running the component. Take a look at the template in the doc directory.

6.4.4 Framework for bash

Bash helper functions for command file parsing, error and log message writing are in the `bash` directory. You should source the `$ANDURIL_HOME/bash/functions.bash` to enable them. There is a template for bash components in the doc directory.

6.4.5 Framework for Python

Import the `anduril` module to access Python framework. To access all the inputs and parameters as variables, have `from anduril.args import *` in your header.

6.4.6 Framework for Lua

The Lua component framework is implemented in `lua/componentskeleton.lua`. The component calls `componentmain` and gives a function as argument. This function performs the main operations of the component and has access to a command file object. Example component:

```
require("componentskeleton")
function execute(cf)
    -- infilename = cf:getinput("inport")
    -- outfilename= cf:getoutput("outport")
    -- intvalue = cf:getparameter("param", "int")
    ...
    return statuscode
end
componentmain(execute)
```

6.5 Component test cases

Component test cases aim to ensure that the component is working correctly in a given environment (see Section 1.4). A component can have any number of test cases. A component with no test cases should not be considered production quality.

The test cases are put into the `testcases` directory located in the component's main directory. The actual test cases are subdirectories of `testcases`. Generally, any name for a testcase directory is allowed, although the following name convention is used: `case1`, `case2`, ..., `casen` for testcases 1, 2, ..., *n* respectively. Each testcase directory contains:

- the `input` subdirectory with the files for the component's input ports;
- the `component.properties` file with the testcase-specific values of the component's parameters;
- either `expected-output` subdirectory or `failure` file. The `expected-output` subdirectory contains files for the expected output of the component, while presence of the file `failure` notifies the engine that the failure of the component on the given input and parameters is expected.

The `input` directory contains a file for each (mandatory) input port; files for optional ports may be missing. The file is named after the input port and may have a file extension corresponding to the port's data type. For example, for a `CSV` port named `table` the `input` directory should contain file `table.csv`.

One may specify the values of parameters of the component for the testcase. Those values should be given in the `component.properties` file in the format `component.`*parameter_name* `=` *parameter_value*, one parameter per line. Moreover, it is possible to specify the timeout for the execution of the component for the given testcase by the parameter `metadata.timeout =` ⟨ *timeout in seconds*⟩. The parameters having a default

value as specified in the `component.xml` file can be omitted from the `component.properties` file. If all the parameters of the component have default values, then the file `component.properties` may be omitted from the testcase directory. Otherwise, all the parameters without default values should be specified in the `component.properties` file. The parameter `metadata.timeout` has the predefined default value, insuring in this way that the execution of the component terminates at some point.

There are two types of test cases: expected success and expected failure. For expected success, the directory `expected-output` contains the output files that are expected to be produced by the component. These files may have file extension, depending on the data types of the output ports. It is not necessary to include files for all output ports in `expected-output`: if an output port is omitted, any file written by the component is accepted. This allows to omit comparisons for binary files (such as PDFs) that are difficult to compare. If `expected-output` is empty, no outputs are checked but the test case still validates that the component can be executed without error with given input.

Some data types like *CSV*, *Latex*, and *TextFile* have special validators (see functionality-class in Section 7) that are used to compare test case outputs to the expected references. For example, the exprected outputs of *TextFile* ports may include regular expressions to represent pieces of single lines. The regular expressions follow the syntax specified in Java standard API and they are encapsulated between {{ and }}. For instance

```
Version {{\d[.]\d\d}} of this code...
```

will match 'Version 1.03 of this code...' and 'Version 4.21 of this code...'. The exact comparison rules of each data type are dependent of the functionality classes in charge.

For expected failure, there must be a file named `failure` present in test case directory. The contents of the file are irrelevant; the file can be empty. For expected failure, the directory `expected-output` must not exist.

6.6 Example component: adding matrices

To demonstrate component implementation, we use a simplified example component, AddMatrix, that adds two or three numeric matrices and adds a constant numeric bias. The descriptor file is in Figure 18 and the R source code in Figure 19. We saw the HTML manual page of AddMatrix earlier in Figure 2 (Section 1.1).

The R source code uses the R package `componentSkeleton`, which is the Anduril component framework for R. The package defines the functions `main`, `NumMatrix.*`, `input.defined`, `get.input`, `get.parameter` and `get.output`, among others. All

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<component>
    <name>AddMatrix</name>
    <version>1.0</version>
    <doc>
    Compute the sum of two or three matrices. Add a constant
    bias to the result.
    </doc>
    <author email="author.email@example.com">Author Name</author>
    <category>arithmetic</category>
    <launcher type="R">
        <argument name="file" value="AddMatrix.r" />
    </launcher>
    <requires>R</requires>
    <inputs>
        <input name="m1" type="Matrix">
            <doc>Input matrix 1.</doc>
        </input>
        <input name="m2" type="Matrix">
            <doc>Input matrix 2.</doc>
        </input>
        <input name="m3" type="Matrix" optional="true">
            <doc>Input matrix 3.</doc>
        </input>
    </inputs>
    <outputs>
        <output name="sum" type="Matrix">
            <doc>Sum of matrices m1, m2 and m3 (if defined),
            plus bias.</doc>
        </output>
    </outputs>
    <parameters>
        <parameter name="bias" type="float" default="0">
            <doc>A bias that is added to all cells of the
            output matrix.</doc>
        </parameter>
    </parameters>
</component>
```

Figure 18: Descriptor file for the example component AddMatrix.

functions are documented in the R manual pages of the package. `NumMatrix.*` are convenience functions for reading and writing matrices. `get.input` and `get.output` provide file names corresponding to input and output ports. `get.parameter` provides the value of a simple parameter and converts it to the relevant R type (here, `float` is converted to a numeric type). `main` parses the command file and calls the given function (here, `execute`) with the command file data structure as an argument. It is the entry point for components.

6.7 Guidelines for designing components

Component development process In the authors' experience, a good strategy for implementing components is based on Test Driven Development, in which component

```
library(componentSkeleton)
execute <- function(cf) {
    m1 <- NumMatrix.read(get.input(cf, 'm1'))
    m2 <- NumMatrix.read(get.input(cf, 'm2'))
    result <- m1+m2
    if (input.defined(cf, 'm3')) {
        m3 <- NumMatrix.read(get.input(cf, 'm3'))
        result <- result + m3
    }
    result <- result + get.parameter(cf, 'bias', 'float')
    NumMatrix.write(get.output(cf, 'sum'), result)
}
main(execute)
```

Figure 19: R source code for the example component AddMatrix.

test cases (Section 6.5) are written before the actual code. New features for existing components can also be implemented using the process.

1. Draft the external interface of the component (`component.xml`).
2. Write one or more test cases. This helps to clarify the interface and functionality of the component. Concrete test cases also enable to easily execute the component (Section 3.2.3).
3. If necessary, refactor the external interface based on insight gained in step 2.
4. Implement component code.
5. Execute test cases and modify code until the tests pass.
6. If necessary, refactor the working code for clarity to ease maintaining. Ensure that tests still pass. Well-tested components can be heavily refactored while maintaining correctness; even the implementation language can be changed if needed.

Designing component collections Adding a few components to a well-established bundle is easy, but designing a substantial component bundle from scratch requires some planning. Components are mostly independent from each other: they are executed in separate processes and they normally do not call each others' code. Components communicate through input and output ports. Thus, interfaces of components should be designed so that they can conveniently be connected together in a workflow. One approach to designing interfaces for a bundle is based on use cases and prototyping.

1. Write down use cases (scenarios) for the novel component bundle. These are high-level descriptions of actual analyses (or workflows).
2. Draft external interfaces for the components, and any novel data types that the bundle needs. This can be done on paper or with concrete `component.xml` files.

3. For each use case, write a prototype workflow that implements the use case using the novel component interfaces. Likewise, this can be done on paper or with concrete AndurilScript files.
4. Iterate steps 2–3 until all use cases can be elegantly satisfied.
5. Implement individual components using, e.g., the process above.
6. The prototype workflows can be converted into workflow test cases by attaching concrete input and expected output files (Section 8.3).

When the bundle is designed by a team, interface specification (steps 1–4) requires collaboration between team members, but components can be implemented by individual developers once their interfaces have been specified.

6.8 Implementing support for new programming languages

This section is for developers who want to implement a component framework for a new language. The goal of a component framework is to make implementing components as convenient as possible. Usually the design of component frameworks is quite similar between languages, so you can get started by studying existing frameworks.

How the component framework is deployed depends on the language platform. As an example, the R framework is an R package that must be installed to the R system before R components can be executed.

A component framework may prodive the following services:

- **Parsing command files**: A minimal parser must be able to parse `input.*`, `output.*` and `parameter.*` lines. The parser should return the parsed contents of the command file in a data structure.
- **Writing to error stream and log stream**: Streams are text files whose names are present in the command file. The format of streams is documented in the built-in data type StringList. Messages may span several lines. Messages are divided by a special divider line.
- **Providing access to input files, output files and parameters**: Necessary information (filenames for input and output files and values for parameters) is present in the command file.
- **Exit status constants**: When a component is finished and exits, it should return an exit status that tells whether the execution was succesful and if not, what kind of error occured. The definitive list of error codes is found in the class ErrorCode in package `fi.helsinki.ltdk.csbl.anduril.component`.
- **I/O support for data types**: The framework should provide read and write support for relevant port data types. The goal is to make file processing convenient.

If the target language provides good native support for a port type, additional support may not be needed. Otherwise, read, write and accessor functionality may be provided. Given a data type (e.g., matrix), the component framework provides a read function that parses a file and returns it as a data structure that is natural to the programming language. For example, in Java, a matrix could be represented as a `double[][]` array, but more complicated data types may need their own class. The framework also provides a write function that writes the data structure to a file, and any accessor functions that are needed to process the data type.

- **Main function**: When a component is executed, the first action is almost always to parse the command file. There are some error conditions that should be handled, such as missing or badly formatted command file. The component framework should provide a main function that performs these tasks. The component calls the main function and receives a command file object that is ready to be used.

7 Defining port data types

Data types for ports are defined using an XML file named `datatypes.xml`. There can be several type definitions in one file. The format of the XML file is defined in Figure 20 and the elements are described in Table 14. Data types may have example files that illustrate the file format. The example files for a data type named `DataType` are located in directory `doc-files/DataType/`. Here, `doc-files` is located in the same directory as `datatypes.xml`. In the XML file, `example-file` elements give a path that is relative to `doc-files/DataType/`.

```
types := type*
type := name version? parent-type? functionality-class? desc?
        presentation-type presentation-desc? example-file*
        extension?
name := STRING
version := STRING
parent-type := STRING
functionality-class := STRING
desc := STRING
presentation-type := "file" | "directory" | "multifile"
presentation-desc := STRING
example-file <path> := STRING
extension := STRING
```

Figure 20: Syntax of descriptor XML files. For details on the notation, see Figure 15.

Element	Description
`name`	Name of the data type.
`version`	Version number, with format `1.0` or `1.0.0` or `1.0.0.0`. If the version number changes, all components in a network that use the old version must be re-executed.
`parent-type`	Name of parent data type, if any.
`functionality-class`	A fully qualified name of a Java class that provides additional support for the data type, such as customized comparison of two files. The class must be a subclass of DataTypeFunctionality.
`desc`	Description of the purpose and logical structure of the data type.
`presentation-type`	One of `file` (single file), `directory` or `multifile` (file with optional auxiliary files with the same basename) .
`presentation-desc`	Description of the physical layout of the files that represent the data type.
`example-file`	An example file. The relative file name is given by the `path` attribute.
`extension`	File extension of the data type, if any.

Table 14: Description of data type specification XML elements.

8 Resource bundles

Components, data types and associated resources are combined into *resource bundles*. The bundle is distributed and installed as one package. The bundle is a directory structure that contains specific files and directories for various resources. See Figure 21 for an overview of bundle structure.

Components and data types can also exist outside a bundle: for example, project-specific private components usually are not part of a bundle. However, all redistributable components should belong to a bundle.

The bundle directory may contain the following files; all but `bundle.xml` are optional:

- `bundle.xml`: A simple description XML file that contains the name and version number of the bundle. See Section 8.1.
- `categories.xml`: Components can be organized into hierarchical categories for the component API by using this simple description XML file. See Section 8.2.
- `datatypes.xml`: Port data types XML file. This contains all data types introduced by the bundle. See Section 7.
- `doc-files`: Files needed by data types, mainly example files. See Section 7.
- `components`: Component repository directory. Components are located in subdi-

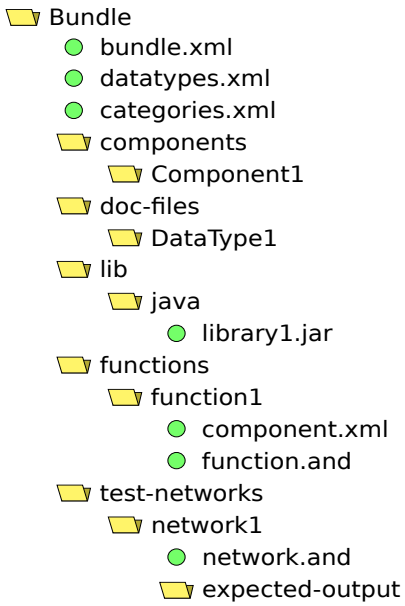


Figure 21: File organization for resource bundles.

rectories of this directory. Component directory structure is desribed in Section 6.

- `lib`: Libraries that are used by components. Libraries are grouped based on the programming language. The main directory `lib` should be empty.
  - `lib/java`: Java libraries: JARs and class files. The directory `lib/java` and JAR files located in it are added to CLASSPATH. Components requiring JAR files from this directory must specify the requirement in `component.xml`.
  - `lib/lua`: Lua libraries, either Lua code or C libraries. This directory is added to `LUA_PATH` and `LUA_CPATH`.
- `functions`: A library of composite component definitions. These definitions are imported whenever a workflow uses the bundle. See Section 8.4 for details.
- `test-networks`: Workflow-level test cases. Each subdirectory of `test-networks` contains one test workflow. See Section 8.3.

8.1 Bundle definition XML files

The bundle definition XML file gives the name of the bundle. The format is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bundle>
    <name>BUNDLENAME</name>
</bundle>
```

8.2 Category definition XML files

The category definition XML file gives the hierachical category structure of the components in the bundle. Upper level (parent) categories are distinguished from lower level (child) categories by defining child categories to have a parent. The format is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<categories>
  <!-- Parent category. -->
  <category>
      <name>Preprocessing</name>
      <doc>Filtering, normalization, quality control.</doc>
  </category>
  <!-- Child category. -->
  <category>
    <name>Normalization</name>
    <doc> Different normalization components and methods </doc>
    <parent-category>Preprocessing</parent-category>
  </category>
</categories>
```

8.3 Workflow-level test cases

Workflow test cases are a part of a resource bundle under the directory `test-networks`. Each test case is a subdirectory of `test-networks`. The test case directory contains the following files:

- `network.and`: workflow configuration file. All output files that are tested for correctness must be directed to `OUTPUT`.
- If the test expects successful execution, `expected-output` contains the expected `output` directory.
- If the test expects failed execution, there must be a file named `failure`. Its contents are not used.
- If the workflow needs any input files, they can be located in the test directory. The workflow configuration file specifies their file names.

If the subdirectory of `test-networks` is named after a component, it is included as a part of the component documentation HTML page.

8.4 Composite component libraries

Common composite component definitions can be stored in the bundle so that whenever a workflow uses the bundle, the composite components are automatically available. Every such composite component is located in the directory `functions/FUNCTION`, where `FUNCTION` is the name of the composite component. This directory contains a component descriptor file `component.xml` that defines the name and interface of the composite component. The composite component definition is located in `function.and`, which is a workflow configuration file containing (at least) one function definition. This function definition has a special format: it does not contain an interface definition but only the body.

Example The descriptor for the composite component `MyFunction` is located in `functions/MyFunction/component.xml`. The descriptor defines input ports `in1` and `in2`; output ports `out1` and `out2`; and parameters `p1` and `p2`. The function body `functions/MyFunction/function.and` is defined as follows:

```
function MyFunction {
    ... processing ...
    result = SomeComponent(in1, in2, x=p1, y=p2)
    return record(out1=result.port1, out2=result.port2)
}
```

9 Integrating Anduril into Java programs

Anduril can be integrated into Java programs using the Java API of Anduril. A typical use case is invoking a workflow from Java. See Figure 22 for an example. Refer to Anduril Java API documentation for details on classes and methods. It is not guaranteed that the Anduril Java API remains unchanged over time, so integration should be re-tested when the Anduril version is updated.

The steps involved in workflow execution are:

1. Loading resource bundles into a repository of components and datatypes (class `Repository`)
2. Configuring a NetworkEvaluator instance by specifying a workflow source from file, string or pre-constructed Network instance
3. Executing the workflow using the NetworkEvaluator instance
4. Accessing output from the execution directory of the engine

After repository loading, workflow parsing and workflow execution, errors must be checked using `hasErrors()`. Errors can be accessed as collections of `StaticError` (repository loading and workflow parsing) or `DynamicError` (workflow execution) using `get*Errors()` methods.

```
import java.io.File;
import fi.helsinki.ltdk.csbl.anduril.core.engine.NetworkEvaluator;
import fi.helsinki.ltdk.csbl.anduril.core.network.Repository;

public class RunWorkflow {
    public static void main(String[] args) throws Exception {
        final File andurilHome = new File("...");
        final File executionDir = new File("execute");
        Repository repository = new Repository(andurilHome);
        repository.load(true, null, null, null);
        if (repository.hasErrors()) {
            /* Handle errors from repository ... */
        }

        File networkSource = new File("workflow.and");
        NetworkEvaluator evaluator = new NetworkEvaluator(executionDir);
        evaluator.setSource(networkSource, repository,
            repository.getSymbolTable());
        evaluator.execute();
        if (evaluator.hasErrors()) {
            /* Handle errors in workflow source or execution */
        }
    }
}
```

Figure 22: Example of invoking Anduril from a Java program.